PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 :
G06F 11/14, 9/46, 17/30

A1

(11) International Publication Number: WO 97/04391

(43) International Publication Date: 6 February 1997 (06.02.97)

(21) International Application Number: PCT/US96/11903

(22) International Filing Date: 18 July 1996 (18.07.96)

(30) Priority Data:
60/001,245 20 July 1995 (20.07.95) US

(60) Parent Application or Grant
(63) Related by Continuation
US 60/001,245 (CIP)
Filed on 20 July 1995 (20.07.95)

(71) Applicant (*for all designated States except US*): NOVELL, INC. [US/US]; 1555 North Technology Way, Orem, UT 84057 (US).

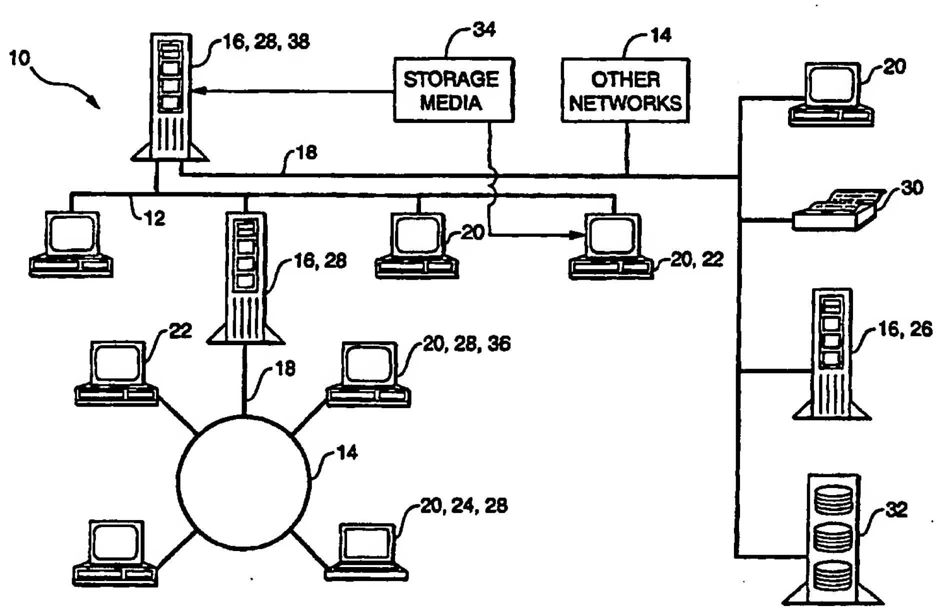
(72) Inventors; and
(75) Inventors/Applicants (*for US only*): FALLS, Patrick, T. [GB/GB]; Meadlands, Broad Layings, Woolton Hill, Newbury, Berkshire RG15 9TT (GB). COLLINS, Brian, J. [GB/GB]; 30 High Drive, New Malden, Surrey KT3 3UG (GB). DRAPER, Stephen, P., W. [GB/GB]; 123 Pack Lane, Basingstoke, Hampshire RG22 5HL (GB).

(74) Agents: OGILVIE, John, W., L. et al.; Computer Law++, Suite 550, 8 East Broadway, Salt Lake City, UT 84111 (US).

(81) Designated States: AL, AM, AT, AU, AZ, BB, BG, BR, BY, CA, CH, CN, CZ, DE, DK, EE, ES, FI, GB, GE, HU, IL, IS, JP, KE, KG, KP, KR, KZ, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, TJ, TM, TR, TT, UA, UG, US, UZ, VN, ARIPO patent (KE, LS, MW, SD, SZ, UG), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG).

Published
With international search report.

(54) Title: TRANSACTION LOG MANAGEMENT IN A DISCONNECTABLE COMPUTER AND NETWORK



(57) Abstract

A method and apparatus are disclosed for managing a transaction log which contains updates representing operations performed on a database replica in a network of disconnectable computers. The invention provides for compression of the log by the identification and removal of redundant updates. Log compression removes apparent inconsistencies between operations performed on disconnected computers, reduces storage requirements on each computer, and speeds up transaction synchronization when the computers are reconnected. The invention also provides for restoration of prior versions of database objects using the log.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AM	Armenia	GB	United Kingdom	MW	Malawi
AT	Austria	GE	Georgia	MX	Mexico
AU	Australia	GN	Guinea	NE	Niger
BB	Barbados	GR	Greece	NL	Netherlands
BE	Belgium	HU	Hungary	NO	Norway
BF	Burkina Faso	IE	Ireland	NZ	New Zealand
BG	Bulgaria	IT	Italy	PL	Poland
BJ	Benin	JP	Japan	PT	Portugal
BR	Brazil	KE	Kenya	RO	Romania
BY	Belarus	KG	Kyrgystan	RU	Russian Federation
CA	Canada	KP	Democratic People's Republic of Korea	SD	Sudan
CF	Central African Republic			SE	Sweden
CG	Congo	KR	Republic of Korea	SG	Singapore
CH	Switzerland	KZ	Kazakhstan	SI	Slovenia
CI	Côte d'Ivoire	LI	Liechtenstein	SK	Slovakia
CM	Cameroon	LK	Sri Lanka	SN	Senegal
CN	China	LR	Liberia	SZ	Swaziland
CS	Czechoslovakia	LT	Lithuania	TD	Chad
CZ	Czech Republic	LU	Luxembourg	TG	Togo
DE	Germany	LV	Latvia	TJ	Tajikistan
DK	Denmark	MC	Monaco	TT	Trinidad and Tobago
EE	Estonia	MD	Republic of Moldova	UA	Ukraine
ES	Spain	MG	Madagascar	UG	Uganda
FI	Finland	ML	Mali	US	United States of America
FR	France	MN	Mongolia	UZ	Uzbekistan
GA	Gabon	MR	Mauritania	VN	Viet Nam

TITLE

TRANSACTION LOG MANAGEMENT IN A DISCONNECTABLE COMPUTER AND NETWORK

INVENTORS

STEPHEN P.W. DRAPER, BRIAN J. COLLINS, AND PATRICK T. FALLS

FIELD OF THE INVENTION

The present invention relates to the management of transaction logs which contain updates representing operations performed on separated disconnectable computers, and more particularly to log compression that is suitable for use with transaction synchronization and with the handling of clashes that may arise during such synchronization.

TECHNICAL BACKGROUND OF THE INVENTION

"Disconnectable" computers are connected to one another only sporadically or at intervals. Familiar examples include "mobile-link" portable computers which are connectable to a computer network by a wireless links and separate server computers in a wide-area network (WAN) or other network. Disconnectable computers can be operated either while connected to one another or while disconnected. During disconnected operation, each computer has its own copy of selected files (or other structures) that may be needed by a user. Use of the selected items may be either direct, as with a document to be edited, or indirect, as with icon files to be displayed in a user interface.

Unfortunately, certain operations performed on the selected item copies may not be compatible or consistent with one another. For instance, one user may modify a file on one computer and another user may delete the "same" file from the other computer. A "synchronization" process may be performed after the computers are reconnected. At a minimum, synchronization attempts to propagate operations performed on one computer to the other computer so that copies of items are consistent with one another.

During synchronization, some disconnectable computers also attempt to detect inconsistencies and to automatically resolve them. These attempts have met with limited success.

For instance, the Coda File System ("Coda") is a client-server system that provides limited support for disconnectable operation. To prepare for disconnection, a user may hoard data in a client cache by providing a prioritized list of files. On disconnection, two copies of each cached file exist: the original stored on the server, and a duplicate stored in the disconnected client's cache. The user may alter the duplicate file, making it inconsistent with the server copy. Upon reconnection, this inconsistency may be detected by comparing timestamps.

However, the inconsistency is detected only if an attempt is made to access one of the copies of the file. The Coda system also assumes that the version stored in the client's cache is the correct version, so situations in which both the original and the duplicate were altered are not properly handled. Moreover, Coda is specifically tailored, not merely to file systems, but to a particular file system (a descendant of the Andrew File System). Coda provides no solution to the more general problem of detecting and resolving inconsistencies in a distributed database that can include objects other than file and directory descriptors.

Various approaches to distributed database replication attempt to ensure consistency between widely separated replicas that collectively form the database. Examples include, without limitation, the replication subsystem in Lotus Notes and the partition synchronization subsystem in Novell NetWare® 4.1 (LOTUS NOTES is a trademark of International Business Machines, Inc. and NETWARE is a registered trademark of Novell, Inc.).

However, some of these approaches to replication are not transactional. Non-transactional approaches may allow partially completed update operations to create inconsistent internal states in network nodes. Non-transactional approaches may also require a synchronization time period that depends directly on the total number of files, directories, or other objects in the replica. This seriously degrades the performance of such approaches when the network connection used

for synchronization is relatively slow, as many modem or WAN links are.

Moreover, in some conventional approaches potentially conflicting changes to a given set of data are handled by simply applying the most recent change and discarding the others. In other conventional systems, users must resolve conflicts with little or no assistance from the system. This can be both tedious and error-prone.

It is well-known in the database arts to maintain a log of transactions. However, conventional disconnectable systems are not traditional database systems. Conventional disconnectable systems lack transaction logs which can be used to identify and then modify or remove certain apparently inconsistent operations to improve the synchronization process. Conventional systems provide no way to compress transaction logs based on the semantics of the logged update operations. Conventional systems also lack a way to use such transaction logs to recreate earlier versions of database objects.

Thus, it would be an advancement in the art to provide a system and method for compressing a log of transactions performed on disconnectable computers.

It would be a further advancement to provide such a system and method which are suited for use with systems and methods for transaction synchronization.

It would also be an advancement to provide such a system and method which are not limited to file system operations but can instead be extended to support a variety of database objects.

Such a system and method are disclosed and claimed herein.

BRIEF SUMMARY OF THE INVENTION

The present invention provides systems and methods for managing a transaction log which represents a sequence of transactions in a network of connectable computers. Each transaction contains at least one update targeting an object in a replica of a distributed target database. The replicas reside on separate computers in the network. In one embodiment the network includes a server computer and a client computer

and a replica of the target database resides on each of the two computers.

One method of the present invention compresses the transaction log by identifying redundant updates and then removing them from the log. Redundant updates are identified by examining the operations performed by the updates and the status of the replicas. The compression is thus based on update semantics, unlike data compression methods such as run-length-encoding which are based only on data values. Semantic tests are also used to identify incompressible update sequences such as file name swaps or sequences that cross a transaction boundary.

A hierarchical log database representing at least a portion of the transaction log assists log management. The log database contains objects corresponding to the updates and transactions in the specified portion of the transaction log. The specified portion of the transaction log may be the entire log, or a smaller portion that only includes recent transactions. The remainder of the transaction log is represented by a compressed linearly accessed log.

Transactions and updates are appended to the log by inserting corresponding objects into the log database. The log database includes an unreplicated attribute or other update history structure. The update history structure is accessed to identify any earlier update referencing an object in the target database that is also referenced by an update in the appended transaction.

The invention also provides other log management capabilities. For instance, each completed transaction in the transaction log has a corresponding transaction sequence number. By specifying a range of one or more transaction sequence numbers, one can retrieve transactions from the transaction log in order according to their respective transaction sequence numbers. In addition, one can locate a desired transaction checkpoint, access the update history structure in the log database, and then construct a prior version of a target database object at it existed at the time represented by that checkpoint.

The present log management invention is suitable for use with various transaction synchronization systems and methods. According to one such, synchronization of the database replicas is performed after the computers are reconnected and includes a "merging out" step, a "merging in" step, and one or more clash handling steps. During the merging out step, operations performed on a first computer are transmitted to a second computer and applied to a replica on the second computer. During the merging in step, operations performed on the second computer are transmitted to the first computer and applied to the first computer's replica.

Some of the clash handling steps detect transient or persistent clashes, while other steps recover from at least some of those clashes. Persistent clashes may occur in the form of unique key clashes, incompatible manipulation clashes, file content clashes, permission clashes, or clashes between the distributed database and an external structure. Recovery may involve insertion of an update before or after a clashing update, alteration of the order in which updates occur, consolidation of two updates into one update, and/or creation of a recovery item. Log compression may be performed as part of clash handling, in preparation for merging, or separately from those procedures.

Transaction synchronization and clash handling are further described in commonly owned copending applications entitled TRANSACTION SYNCHRONIZATION IN A DISCONNECTABLE COMPUTER AND NETWORK and TRANSACTION CLASH MANAGEMENT IN A DISCONNECTABLE COMPUTER AND NETWORK, filed the same day and having the same inventors as the present application.

The features and advantages of the present invention will become more fully apparent through the following description and appended claims taken in conjunction with the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

To illustrate the manner in which the advantages and features of the invention are obtained, a more particular description of the invention summarized above will be rendered by reference to the appended drawings. Understanding that

these drawings only provide selected embodiments of the invention and are not therefore to be considered limiting of its scope, the invention will be described and explained with additional specificity and detail through the use of the accompanying drawings in which:

Figure 1 is a schematic illustration of a computer network suitable for use with the present invention.

Figure 2 is a diagram illustrating two computers in a network, each configured with a database manager, replica manager, network link manager, and other components according to the present invention.

Figure 3 is a diagram further illustrating the replica managers shown in Figure 2.

Figure 4 is a flowchart illustrating log management methods of the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

Reference is now made to the Figures wherein like parts are referred to by like numerals. The present invention relates to a system and method which facilitate disconnected computing with a computer network. One of the many computer networks suited for use with the present invention is indicated generally at 10 in Figure 1.

In one embodiment, the network 10 includes Novell NetWare® network operating system software, version 4.x (NETWARE is a registered trademark of Novell, Inc.). In alternative embodiments, the network includes Personal NetWare, NetWare Mobile, VINES, Windows NT, LAN Manager, or LANtastic network operating system software (VINES is a trademark of Banyan Systems; NT and LAN Manager are trademarks of Microsoft Corporation; LANtastic is a trademark of Artisoft). The network 10 may include a local area network 12 which is connectable to other networks 14, including other LANs, wide area networks, or portions of the Internet, through a gateway or similar mechanism.

The network 10 includes several servers 16 that are connected by network signal lines 18 to one or more network clients 20. The servers 16 may be file servers, print servers, database servers, Novell Directory Services servers, or a

combination thereof. The servers 16 and the network clients 20 may be configured by those of skill in the art in a wide variety of ways to operate according to the present invention.

The network clients 20 include personal computers 22, laptops 24, and workstations 26. The servers 16 and the network clients 20 are collectively denoted herein as computers 28. Suitable computers 28 also include palmtops, notebooks, personal digital assistants, desktop, tower, micro-, mini-, and mainframe computers. The signal lines 18 may include twisted pair, coaxial, or optical fiber cables, telephone lines, satellites, microwave relays, modulated AC power lines, and other data transmission means known to those of skill in the art.

In addition to the computers 28, a printer 30 and an array of disks 32 are also attached to the illustrated network 10. Although particular individual and network computer systems and components are shown, those of skill in the art will appreciate that the present invention also works with a variety of other networks and computers.

At least some of the computers 28 are capable of using floppy drives, tape drives, optical drives or other means to read a storage medium 34. A suitable storage medium 34 includes a magnetic, optical, or other computer-readable storage device having a specific physical substrate configuration. Suitable storage devices include floppy disks, hard disks, tape, CD-ROMs, PROMs, RAM, and other computer system storage devices. The substrate configuration represents data and instructions which cause the computer system to operate in a specific and predefined manner as described herein. Thus, the medium 34 tangibly embodies a program, functions, and/or instructions that are executable by at least two of the computers 28 to perform log management steps of the present invention substantially as described herein.

With reference to Figure 2, at least two of the computers 28 are disconnectable computers 40 configured according to the present invention. Each disconnectable computer 40 includes a database manager 42 which provides a location-independent interface to a distributed hierarchical target database embodied in convergently consistent replicas 56. Suitable

databases include Novell directory services databases supported by NetWare 4.x.

A database is a collection of related objects. Each object has associated attributes, and each attribute assumes one or more values at any given time. Special values are used internally to represent NULL, NIL, EMPTY, UNKNOWN, and similar values. Each object is identified by at least one "key." Some keys are "global" in that they are normally unique within the entire database; other keys are "local" and are unique only within a proper subset of the database. A database is "hierarchical" if the objects are related by their relative position in a hierarchy, such as a file system hierarchy. Hierarchies are often represented by tree structures.

The target database includes file descriptor objects, directory descriptor objects, directory services objects, printer job objects, or other objects. The target database is distributed in that entries are kept in the replicas 56 on different computers 40. Each replica 56 in the target database contains at least some of the same variables or records as the other replicas 56. The values stored in different replicas 56 for a given attribute are called "corresponding values." In general, corresponding values will be equal.

However, replicas 56 at different locations (namely, on separate computers 40) may temporarily contain different values for the same variable or record. Such inconsistencies are temporary because changes in value are propagated throughout the replicas 56 by the invention. Thus, if the changes to a particular variable or record are infrequent relative to the propagation delay, then all replicas 56 will converge until they contain the same value for that variable or record.

More generally, the present invention provides a basis for a family of distributed software applications utilizing the target database by providing capabilities which support replication, distribution, and disconnectability. In one embodiment, the database manager 42 includes one or more agents 44, such as a File Agent, a Queue Agent, or a Hierarchy Agent. The database manager 42 hides the complexity of distribution of data from the application programs. Distributed programs make

requests of the database manager 42, which dispatches each request to an appropriate agent 44.

Each agent 44 embodies semantic knowledge of an aspect or set of objects in the distributed target database. Under this modular approach, new agents 44 can be added to support new distributed services. For instance, assumptions and optimizations based on the semantics of the hierarchy of the NetWare File System are embedded in a Hierarchy Agent, while corresponding information about file semantics are embedded in a File Agent. In one embodiment, such semantic information is captured in files defining a schema 84 (Figure 3) for use by agents 44.

The schema 84 includes a set of "attribute syntax" definitions, a set of "attribute" definitions, and a set of "object class" (also known as "class") definitions. Each attribute syntax in the schema 84 is specified by an attribute syntax name and the kind and/or range of values that can be assigned to attributes of the given attribute syntax type. Attribute syntaxes thus correspond roughly to data types such as integer, float, string, or Boolean in conventional programming languages.

Each attribute in the schema 84 has certain information associated with it. Each attribute has an attribute name and an attribute syntax type. The attribute name identifies the attribute, while the attribute syntax limits the values that are assumed by the attribute.

Each object class in the schema 84 also has certain information associated with it. Each class has a name which identifies this class, a set of super classes that identifies the other classes from which this class inherits attributes, and a set of containment classes that identifies the classes permitted to contain instances of this class.

An object is an instance of an object class. The target database contains objects that are defined according to the schema 84 and the particulars of the network 10. Some of these objects may represent resources of the network 10. The target database is a "hierarchical" database because the objects in the database are connected in a hierarchical tree structure. Objects in the tree that can contain other objects are called

"container objects" and must be instances of a container object class.

A specific schema for the Hierarchy Agent will now be described; other agents may be defined similarly. The ndr_dodb_server class is the top level of the HA-specific database hierarchy. Since a database may contain many servers, the name is treated as a unique key for HA servers within a database.

```
CLASS      ha_server
{
     SUPERCLASS      ndr_dodb_object_header;
     PARENT          ndr_dodb_database;
     PROPERTY        NDR_OS_CLASS_FLAG_FULLY_REPLICATED;
     ATTRIBUTE
     {
          ha_server_name      server_name
          PROPERTY            NDR_OS_ATTR_FLAG_SIBLING_KEY;
     }
}
CONSTANT   HA_VOLUME_NAME_MAX = 32;
DATATYPE   ha_volume_name      STRING HA_VOLUME_NAME_MAX;
DATATYPE   ha_volume_id        BYTE;
```

A volume has a name, which must be unique within the server and can be used as the root component of a path name:

```
CLASS      ha_volume
{
     SUPERCLASS      ndr_dodb_object_header;
     PARENT          ha_server;
     PROPERTY        NDR_OS_CLASS_FLAG_NAMESPACE_ROOT;
     ATTRIBUTE
     {
          ha_volume_name      volume_name
          PROPERTY            NDR_OS_ATTR_FLAG_SIBLING_KEY |
                              NDR_OS_ATTR_FLAG_IS_DOS_FILENAME;
          ha_volume_id        volume_id;
     }
}
```

In order to allocate unique volume identifiers this object holds the next free volume ID. Initially this is set to 1, so that the SYS volume can be given ID 0 when it is added to the database, in case any applications make assumptions about SYS:

```
CLASS      ha_next_volume
{
     SUPERCLASS      ndr_dodb_object_header;
     PARENT          ha_server;
     PROPERTY        NDR_OS_CLASS_FLAG_UNREPLICATED;
```

```
        ATTRIBUTE
        {
            ndr_dodb_dummy_key  dummy_key
            PROPERTY            NDR_OS_ATTR_FLAG_SIBLING_KEY
            COMPARISON          ndr_dodb_dummy_key_compare
            VALIDATION          ndr_dodb_dummy_key_validate;
            ha_volume_id        next_free_volume_id;
        }
}
```

A file or directory name can be 12 (2-byte) characters long:

```
CONSTANT  HA_FILENAME_MAX = 24;
DATATYPE  ha_filename          STRING HA_FILENAME_MAX;
```

The ha_file_or_dir_id is a compound unique key embracing the file or directory ID that is allocated by the server, as well as the server-generated volume number. The latter is passed as a byte from class 87 NetWare Core Protocols from which it is read directly into vol (declared as a byte below). Elsewhere in the code the type ndr_host_volume_id (a UINT16) is used for the same value.

```
DATATYPE    ha_file_or_dir_id
{
    ULONG           file_or_dir;
    ha_volume_id    vol;
}
```

Files and directories have many shared attributes, the most important being the file name. This must be unique for any parent directory.

```
CLASS     ha_file_or_dir
{
    PARENT          ha_directory;
    SUPERCLASS      ndr_dodb_object_header;
    ATTRIBUTE
    {
        ha_filename         filename
        PROPERTY            NDR_OS_ATTR_FLAG_SIBLING_KEY |
                            NDR_OS_ATTR_FLAG_IS_DOS_FILENAME;
        ha_file_or_dir_id   id
        PROPERTY            NDR_OS_ATTR_FLAG_GLOBAL_KEY |
                            NDR_OS_ATTR_FLAG_UNREPLICATED

        GROUP               file_or_dir_id_group;
        ULONG               attributes;
        SHORT               creation_date;
```

```
            SHORT                creation_time;
            ndr_dodb_auth_id     creation_id;
            SHORT                access_date;
            SHORT                archive_date;
            SHORT                archive_time;
            ndr_dodb_auth_id     archive_id;
        }
}
```

A file has some additional attributes not present in a directory, and may contain a contents fork which can be accessed via a file distributor 90 (Figure 3):

```
CLASS     ha_file
{
        SUPERCLASS      ha_file_or_dir;
        PROPERTY        NDR_OS_CLASS_FLAG_DEFINE_REPLICAS |
                  NDR_OS_CLASS_FLAG_HAS_PARTIALLY_REPLICATED_FILE
|
                        NDR_OS_CLASS_FLAG_HAS_FILE_PATH_NAME |
                        NDR_OS_CLASS_FLAG_PARENT_HAS_RSC;
        ATTRIBUTE
        {
            BYTE                    execute_type;
            SHORT                   update_date
            property                NDR_OS_ATTR_FLAG_UNREPLICATED;
            SHORT                   update_time
            property                NDR_OS_ATTR_FLAG_UNREPLICATED;
            ndr_dodb_auth_id        update_id
            property                NDR_OS_ATTR_FLAG_UNREPLICATED;
            ULONG                   length
            property                NDR_OS_ATTR_FLAG_UNREPLICATED;
        }
}
```

A directory does not possess a contents fork for file distributor 90 access. The access rights mask is inherited and should be managed by like access control lists ("ACLs"):

```
CLASS     ha_directory
{
        SUPERCLASS      ha_file_or_dir;
        PROPERTY        NDR_OS_CLASS_FLAG_DEFINE_REPLICAS |
                        NDR_OS_CLASS_FLAG_HAS_FILE_PATH_NAME |
                        NDR_OS_CLASS_FLAG_HAS_RSC;
                        //replication support count
        ATTRIBUTE
        {
            BYTE                    access_rights_mask;
            SHORT                   update_date;
            SHORT                   update_time;
            ndr_dodb_auth_id        update_id;
            SHORT                   rsc
            PROPERTY                NDR_OS_ATTR_FLAG_IS_RSC |
```

```
                              NDR_OS_ATTR_FLAG_UNREPLICATED;
    }
}
```

The root directory must appear at the top of the hierarchy below the volume. Its name is not used; the volume name is used instead. This is the top of the replication hierarchy and therefore is the top level RSC in this hierarchy:

```
CLASS       ha_root_directory
{
    SUPERCLASS      ha_directory;
    PARENT          ha_volume;
    PROPERTY        NDR_OS_CLASS_FLAG_DEFINE_REPLICAS |
                    NDR_OS_CLASS_FLAG_HAS_RSC;
}
```

In one embodiment, schemas such as the schema 84 are defined in a source code format and then compiled to generate C language header files and tables. The named source file is read as a stream of lexical tokens and parsed using a recursive descent parser for a simple LL(1) syntax. Parsing an INCLUDE statement causes the included file to be read at that point. Once a full parse tree has been built (using binary nodes), the tree is walked to check for naming completeness. The tree is next walked in three passes to generate C header (.H) files for each included schema file. The header generation passes also compute information (sizes, offsets, and so forth) about the schema which is stored in Id nodes in the tree. Finally, the complete tree is walked in multiple passes to generate the schema table C source file, which is then ready for compiling and linking into an agent's executable program.

Each disconnectable computer 40 also includes a replica manager 46 which initiates and tracks location-specific updates as necessary in response to database manager 42 requests. The

replica manager is discussed in detail in connection with later Figures.

A file system interface 48 on each computer 40 mediates between the replica manager 46 and a storage device and controller 54. Suitable file system interfaces 48 include well-known interfaces 48 such as the File Allocation Table ("FAT") interfaces of various versions of the MS-DOS® operating system (MS-DOS is a registered trademark of Microsoft Corporation), the XENIX® file system (registered trademark of Microsoft Corporation), the various NOVELL file systems (trademark of Novell, Inc.), the various UNIX file systems (trademark of Santa Cruz Operations), the PCIX file system, the High Performance File System ("HPFS") used by the OS/2 operating system (OS/2 is a mark of International Business Machines Corporation), and other conventional file systems.

Suitable storage devices and respective controllers 54 include devices and controllers for the media disclosed above in connection with the storage medium 34 (Figure 1) and other conventional devices and controllers, including non-volatile storage devices. It is understood, however, that the database replicas 56 stored on these media are not necessarily conventional even though the associated devices and controllers 54 may themselves be known in the art.

Each computer 40 also has a network link manager 50 that is capable of establishing a network connection 52 with another disconnectable computer 40. Suitable network link managers 50 include those capable of providing remote procedure calls or an equivalent communications and control capability. One embodiment utilizes "DataTalk" remote procedure call software

with extended NetWare Core Protocol calls and provides functionality according to the following interface:

rpc_init()	Initialize RPC subsystem
rpc_shutdown()	Shutdown RPC subsystem
rpc_execute()	Execute request at single location
rpc_ping()	Ping a location (testing)
rpc_claim_next_execute()	Wait until the next rpc_execute() is guaranteed to be used by this thread
rpc_free_next_execute()	Allow others to use rpc_execute()

Those of skill in the art will appreciate that other remote procedure call mechanisms may also be employed according to the present invention. Suitable network connections 52 may be established using packet-based, serial, internet-compatible, local area, metropolitan area, wide area, and wireless network transmission systems and methods.

Figures 2 and 3 illustrate one embodiment of the replica manager 46 of the present invention. A replica distributor 70 insulates the database manager 42 from the complexities caused by having database entries stored in replicas 56 on multiple computers 40 while still allowing the database manager 42 to efficiently access and manipulate individual database objects, variables, and/or records. A replica processor 72 maintains information about the location and status of each replica 56 and ensures that the replicas 56 tend to converge.

A consistency distributor 74 and a consistency processor 76 cooperate to maintain convergent and transactional consistency of the database replicas 56. The major processes used include an update process which determines how transaction updates are applied, an asynchronous synchronization process that asynchronously synchronizes other locations in a location set, a synchronous synchronization process that synchronously

forces two locations into sync with each other, an optional concurrency process that controls distributed locking, and a merge process that adds new locations to a location set. In one embodiment, processes for synchronization and merging are implemented using background software processes with threads or similar means. The concurrency process may be replaced by a combination of retries and clash handling to reduce implementation cost and complexity.

Each location is identified by a unique location identifier. A "location sync group" is the group of all locations that a specific location synchronizes with. The location sync group for a database replica 56 on a client 20 is the client and the server 16 or other computer 28 that holds a master replica 56; the computer 28 holding the master replica 56 is the "storage location" of the target database. The location sync group for the computer 28 that holds the master replica 56 is all computers 28 connectable to the network that hold a replica 56. A "location set" is a set of presently connected locations in a location sync group. Locations in an "active location set" have substantially converged, while those in a "merge location set" are currently being merged into the active location set. Objects are read at a "reference location" and updated at an "update location," both of which are local when possible for performance reasons. To support concurrency control, objects require a "lock location" where they are locked for read or update; the local location is the same for all processes in a given location set.

According to one update process of the present invention, the updates for a single transaction are all executed at one

update location. Each group of updates associated with a single transaction have a processor transaction identifier ("PTID") containing the location identifier of the update location and a transaction sequence number. The transaction sequence number is preferably monotonically consecutively increasing for all completed transactions at a given location, even across computer 28 restarts, so that other locations receiving updates can detect missed updates.

The PTID is included in update details written to an update log by an object processor 86. An update log (sometimes called an "update stream") is a chronological record of operations on the database replica 56. Although it may be prudent to keep a copy of an update log on a non-volatile storage device, this is not required. The operations will vary according to the nature of the database, but typical operations include adding objects, removing objects, modifying the values associated with an object attribute, modifying the attributes associated with an object, and moving objects relative to one another.

The PTID is also included as an attribute of each target database object to reflect the latest modification of the object. In one embodiment, the PTID is also used to create a unique (within the target database) unique object identifier ("UOID") when a target database object is first created.

A target database object may contain attributes that can be independently updated. For instance, one user may set an archive attribute on a file while a second user independently renames the file. In such situations, an object schema 84 may define attribute groups. A separate PTID is maintained in the

object for each attribute group, thereby allowing independent updates affecting different attribute groups of an object to be automatically merged without the updates being treated as a clash.

The consistency distributor 74 gathers all of the updates for a single transaction and sends them, at close transaction time, to the update location for the transaction. The consistency processor 76 on the update location writes the updates to a transaction logger 88. In one embodiment, the transaction logger 88 buffers the updates in memory (e.g. RAM). If the update location is not local then the updates are committed to the transaction log and the PTID for the transaction is returned, so that the same updates can be buffered locally; this allows all updates to be applied in order locally. In this manner the transaction updates are applied to the update location.

An objective of one asynchronous synchronization process of the present invention is to keep the rest of the locations in the location set in sync without unacceptable impact on foreground software process performance. This is achieved by minimizing network transfers.

A process of the consistency processor 76 (such as a background software process) either periodically or on demand requests the transaction logger 88 to force write all pending transactions to the log and (eventually) to the target database. The consistency processor 76 also causes the batch of updates executed at an update location to be transmitted to all other locations in the current location set as a "SyncUpdate" request. These updates are force written to the

log before they are transmitted to other locations, thereby avoiding use of the same transaction sequence number for different transactions in the event of a crash.

The SyncUpdate requests are received by other locations in the same location set and applied to their in-memory transaction logs by their respective consistency processors 76. Each consistency processor 76 only applies SyncUpdate transactions which have sequence numbers that correspond to the next sequence number for the specified location.

The consistency processor 76 can determine if it has missed updates or received them out of order by examining the PTID. If updates are missed, the PTID of the last transaction properly received is sent to the consistency distributor 74 that sent out the updates, which then arranges to send the missing updates to whichever consistency processors 76 need them.

Acknowledged requests using threads or a similar mechanism can be used in place of unacknowledged requests sent by non-central locations. Non-central locations (those not holding a master replica 56) only need to synchronize with one location and thus only require a small number of threads. To promote scalability, however, central locations preferably use unacknowledged broadcasts to efficiently transmit their SyncUpdate requests.

The asynchronous synchronization process causes SyncUpdate requests to be batched to minimize network transfers. However, the cost paid is timeliness. Accordingly, a synchronous synchronization process according to the present invention may be utilized to selectively speed up synchronization. The

synchronous synchronization process provides a SyncUptoPTID request and response mechanism.

In one embodiment, the SyncUptoPTID mechanism utilizes a SyncState structure which is maintained as part of a location state structure or location list that is managed by a location state processor 80 in the memory of each computer 28. The SyncState structure for a given location contains a location identifier and corresponding transaction sequence number for the most recent successful transaction applied from that location. The SyncState structure is initialized from the update log at startup time and updated in memory as new transactions are applied.

A SyncUptoPTID request asks a destination to bring itself up to date with a source location according to a PTID. The destination sends a copy of the SyncState structure for the source location to that source location. The source location then sends SyncUpdate requests to the destination location, as previously described, up to an including the request with the PTID that was specified in the SyncUptoPTID request. In a preferred embodiment, the central server is a NetWare server and the SyncUptoPTID requirements are approximately 100 bytes per location, so scalability is not a significant problem for most systems.

A merge process according to the present invention includes merging location sets when disconnected disconnectable computers are first connected or reconnected. For instance, merging location sets normally occurs when a computer new to the network starts up and merges into an existing location set.

Merging can also happen when two sets of computers become connected, such as when a router starts.

Merging occurs when two replicas 56 are resynchronized after the computers 28 on which the replicas 56 reside are reconnected following a period of disconnection. Either or both of the computers 28 may have been shut down during the disconnection. A set of updates are "merged atomically" if they are merged transactionally on an all-or-nothing basis. A distributed database is "centrally synchronized" if one computer 28, sometimes denoted the "central server," carries a "master replica" with which all merges are performed.

Portions of the master replica or portions of another replica 56 may be "shadowed" during a merge. A shadow replica, sometimes called a "shadow database", is a temporary copy of at least a portion of the database. The shadow database is used as a workspace until it can be determined whether changes made in the workspace are consistent and thus can all be made in the shadowed replica, or are inconsistent and so must all be discarded. The shadow database uses an "orthogonal name space." That is, names used in the shadow database follow a naming convention which guarantees that they will never be confused with names in the shadowed database.

A "state-based" approach to merging compares the final state of two replicas 56 and modifies one or both replicas 56 to make corresponding values equal. A "log-based" or "transaction-based" approach to merging incrementally applies successive updates made on a first computer 28 to the replica 56 stored on a second computer 28, and then repeats the process with the first computer's replica 56 and the second computer's

update log. A hybrid approach uses state comparison to generate an update stream that is then applied incrementally. The present invention preferably utilizes transaction-based merging rather than state-based or hybrid merging.

As an illustration, consider the process of merging a single new location A with a location set containing locations B and C. In one embodiment, the following performance goals are satisfied:

(a) Use of locations B and C is not substantially interrupted by synchronization of the out-of-date location A with B and C; and
(b) Users connected to location A (possibly including multiple users if location B is a gateway) are able to see the contents of the other locations in the set within a reasonable period of time.

Merging typically occurs in three phases. During a "merging out" phase location A sends newer updates to location B. For instance, if A's location list contains PTID 50:14 (location identifier:transaction sequence number) and B's location list contains PTID 50:10, then the newer updates sent would correspond to PTID values 50:11 through 50:14.

During a "merging in" phase new updates in the merge location B are merged into A's location. For instance, suppose A's location list contains PTIDs 100:12 and 150:13 and B's location list contains PTIDs 100:18 and 150:13. Then the new updates would correspond to PTID values 100:13 through 100:18. If updates are in progress when merging is attempted, the initial attempt to merge will not fully succeed, and additional iterations of the merging in and merging out steps are performed.

In one embodiment, merging does not include file contents synchronization. Instead file contents are merged later,

either by a background process or on demand triggered by file access. This reduces the time required for merging and promotes satisfaction of the two performance goals identified above. In embodiments tailored to "slow" links, merging is preferably on-going to take advantage of whatever bandwidth is available without substantially degrading the perceived performance of other processes running on the disconnectable computers.

In embodiments employing an update log, the log is preferably compressed prior to merging. Compression reduces the number of operations stored in the log. Compression may involve removing updates from the log, altering the parameters associated with an operation in a given update, and/or changing the order in which updates are stored in the log.

In one embodiment, all Object Database calls come through the consistency distributor 74, which manages distributed transaction processing and maintains consistency between locations. Almost all calls from a location distributor 78 are made via the consistency distributor 74 because the consistency distributor 74 supports a consistent view of the locations and the database replicas 56 on them.

The consistency distributor 74 and an object distributor 82 support multiple concurrent transactions. This is needed internally to allow background threads to be concurrently executing synchronization updates. It could also be used to support multiple concurrent gateway users. In an alternative embodiment, multiple concurrent transactions on the same session is supported through the consistency distributor 74.

In one embodiment, the consistency distributor 74 and the consistency processor 76 are implemented in the C programming language as a set of files which provide the functionality described here. Files CD.H and CD.C implement part of the consistency distributor 74. A separate module having files CD_BG.H and CD_BG.C is responsible for background processes associated with merging and synchronization. A module having files CDI.H and CDI.C contains functions used by both the CD and CD_BG modules. These modules provide functionality according to the following interface:

cd_init	Init CD
cd_shutdown	Shutdown CD
cd_create_replica	Create a replica of a specified database
cd_remove_replica	Remove a replica of a specified database
cd_load_db	Load an existing database
cd_unload_db	Unload an existing database
cd_merge_start	Start merge of active and merge location sets
cd_merge_stop	Stop merge
cd_start_txn	Start a CD transaction
cd_set_txn_ref_loc	Set reference/update lid (location identifier) for txn (transaction)
cd_get_txn_desc	Get a txn descriptor given a txn id
cd_abort_txn	Abort a CD transaction
cd_end_txn	End a CD transaction
cd_commit	Commit all previously closed txns to disk
cd_execute_txn	Execute locks and updates for a txn
cd_read	Do read or lookup request
cd_readn	Do readn
cd_lookup_by_uoid	Do lookup using UOID
cd_add_lock	Add an object or agent lock
cd_remove_lock	Remove an object or agent lock
cd_modify_attribute	Modify a single attribute in a previously read object
cd_init_new_doid	Setup all fields in a new doid
cd_add	Add a new object
cd_remove	Remove an object
cd_move	Move an object
cd_set_marker	Add marker point to txn
cd_revert_to_marker	Revert txn state to last marker
cd_get_effective_access_right	

	Get the effective access rights for the current session and object
cd_convert_uoid2doid	Convert UOID to DOID
cd_sync_object	Get the server to send a newly replicated object
cd_bg_init	Initialize CD background processes
cd_bg_merge	Execute a background merge
cd_bg_sync_remote_upto_ptid	Bring remote location up to date with local PTID
cdi_init	
cdi_shutdown	
cdi_execute_ack_sys	Execute acknowledged request using system session
cdi_execute_ack	Execute acknowledged request
cdi_apply_locks	Apply locks for txn
cdi_abort_prc_txn	Remove all locks already set for a txn
//Forced update location (used to change update location when executing clash handler functions)	
cdi_register_forced_update_location	Register location to be used as update location for thread
cdi_unregister_forced_update_location	Unregister location to be used as update location for thread
cdi_get_forced_update_location	Get forced update location for thread
cdi_sync_upto_ptid	Bring location up to date with PTID
cdi_sync_upto_now	Bring location up to date with latest PTID
cdi_sync_loc_list	Make my location list consistent with destination location list and return info on mismatch of PTIDs
cdi_read_loc_list	Read location list
cdi_sync_upto_dtid	Bring location up to date with DTID

Since updates are cached during a transaction, special handling of reads performed when updates are cached is required. In one embodiment, the caller of cd_read() or cd_readn() sees the results of all updates previously executed in the transaction. In an alternative embodiment, for cd_read() reads will see all previously added objects and will see the modified attributes of objects, but will not see the effects of moves or removes. Thus if an object is removed

during a transaction the read will behave as if it has not been removed. The same is true for moved objects. Modifications to keys will have no effect on reads using the keys. The cd_readn() function behaves as if none of the updates in the current transaction have been applied.

In one embodiment, the consistency processor 76, which processes all distributed object database requests, includes background processes that manage object database updates on local locations and synchronization of locations. Within this embodiment, a CP module contains a dispatcher for all requests which call functions that have a prefix of "cpXX_"; a CPR module processes read requests; a CPU module processes update and lock requests; a CPSM module processes synchronization and merging requests; a CP_BG module controls background processing which includes scheduling multiple background threads, controlling the state of all local locations and synchronization of local locations with local and remote locations; and a CPUI module provides functions that are shared by the CP_BG and CPx modules. These modules provide functionality according to the following interface:

cp_init	Includes performing mounting of local locations and recovery of TL (transaction logger 88) and OP (object processor 86)
cp_shutdown	Shutdown CP
cp_process	Process a consistency request
cp_clear_stats	Reset CP statistics
cp_dump_stats	Dump CP statistics to the log
cpr_process_read	Process OP read or lookup request
cpr_process_readn	Process readn request
cpu_register_dtid	Register use of a DTID at a reference location
cpu_execute_txn	Execute single txn at reference location
cpu_commit	Commit all txns for session
cpu_add_locks	Add list of locks
cpu_remove_locks	Remove list of locks

cpu_abort_prc_txn	Remove object locks for specified transaction
cpsm_sync_upto_ptid	Bring remote locations up to date as far as given PTID
cpsm_get_latest_ptid	Obtain the latest PTID
cpsm_get_sync_object	Remote machine wants to sync a newly replicated object
cpsm_sync_object	Add a newly replicated object to the local database
cpsm_get_sync_update	Get a local sync_update
cpsm_sync_update	Apply multiple update txns to location
cpsm_read_loc_list	Read list of locations and states
cpsm_sync_loc_list	Sync location list
cpsm_merge_loc_list	Attempt to merge my location list with other location list
cpsm_sync_finished	Remote machine is notifying us that a sync_upto_ptid has completed
cpsm_request_merge	Request a merge of this location with the central server
cpui_init	Initialize internal structures
cpui_shutdown	Shutdown CPUI subsystem
cpui_execute_txn	Execute update txn at a local location
cpui_apply_update_list_to_db	Apply an update list to an OP database
cpui_commit	Commit all txns at location
cpui_flush	Flush all txns to object database at location
cpui_replay_logged_transactions	Replay transactions from the log that have not been committed to OP
cp_bg_init	Initialize CP_BG subsystem
cp_bg_shutdown	Shutdown CP_BG subsystem
cp_bg_handle_distributed_request	Handle a request that requires remote communication
cp_bg_notify_close_txn	Notify CP_BG of a closed transaction
cp_bg_notify_commit	Notify CP_BG that all txns are committed at a location
cp_bg_attempt_send_flush	Attempt to send out and flush txns
cp_bg_notify_load	Notify CP_BG of a newly loaded DB
cp_bg_notify_unload	Notify CP_BG of a newly unloaded DB
cp_bg_flush_upto_ptid	Force all transactions upto the specified ptid to the migrated state

The location distributor 78 in each replica manager 46 and the location state processor 80 are used to determine the storage locations of database entries. In one embodiment, the

location state processor 80 uses a cache of the current state of locations and maintains state information on the merging process. The location state processor 80 is responsible for processing remote requests which pertain to the location list.

All locations that are up at any time within a sync group are in either the ACTIVE or MERGE location sets. The ACTIVE location set contains all locations that are in sync with the local location up to certain sync watermarks. The MERGE location set contains all nodes that are not in sync with the local location, either through not having updates the active set does have, or through having updates the active set does not have.

Locations in the MERGE set enter the ACTIVE set through the two-way merging process described above, under control of the consistency distributor 74 and the consistency processor 76. Once in the ACTIVE set, a location should never leave it until the location goes down.

Each location continuously sends out its local updates to other members of its active location set as part of the merging process. The PTID in a location's log that was last sent out in this manner is called the location's "low watermark" PTID. For a location to enter the active set it must have all PTIDS in its local log up to the low watermark PTID; only the merging process used to move a location from the MERGE to the ACTIVE location set is capable of propagating early transactions. Each location also maintains a "high watermark" PTID which is the last transaction (in local log order) that has been committed, and is thus a candidate for sending out in a background sync update.

The replica managers 46 track the last transaction sequence number made by every location up to the low watermark PTID in order to know whether a location is up to date with another location's low watermark. The log ordering may be different in different locations, up to an interleave.

One embodiment of the location state processor 80 provides functionality according to the following interface:

ls_init	Initialize LS
ls_shutdown	Shutdown LS
ls_close_db	Clear out all entries for a database
ls_allocate_new_lid	Allocate a new location identifier for use by a new replica
ls_add	Add a new location
ls_remove	Remove a location
ls_modify_local_tid	Modify a location entry's local transaction identifier (sequence number)
ls_modify_state	Modify a location entry's state
ls_get_loc_list	Get list of locations
ls_get_loc_sync_list	Get list of locations for syncing
ls_get_next_loc	Get next location
ls_get_first_in_loc_list	Get first location in list that is in current location set
ls_get_loc_entry	Get location entry given lid (location identifier)
ls_get_first_ref_loc	Get nearest reference location in provided list
ls_get_first_ref_loc_in_list	Get first reference location in provided list
ls_get_lock_loc	Get lock location for location set
ls_higher_priority	Determine which location has highest priority
ls_complete_merge	Complete the merge process
ls_set_sync_watermarks	Set the high and low watermark PTIDs used in syncing and merging

The object distributor 82 manages ACLs and otherwise manages access to objects in the database. In one embodiment, the object distributor 82 provides functionality according to this interface:

```
typedef void*  ndr_od_db_handle;   //open database handle
//lint -strong(AJX,ndr_od_txn_id)
```

```
//object distributor transaction instance identifier
typedef void*  ndr_od_txn_id;
#define   NDR_OD_INVALID_TXN_ID    (ndr_od_txn_id)0
typedef struct //Txn info returned by NdrOdGetTxnInfo
{
     ndr_od_db_handle          db;        /* database     */
     ndr_dodb_session_type     session;   /* session      */
}  ndr_od_txn_info;

//Start a new clash txn for this session
ndr_ret EXPORT
NdrOdStartClashTxn(
     ndr_od_db_handle      db_handle,
     /* -> Handle to the open DB */
     ndr_dodb_session_type    session, /* -> session     */
     ndr_od_txn_id           *txn_id);   /* <- txn id     */

//Find out what databases are available
ndr_ret EXPORT
NdrOdEnumerateDBs(
     ndr_od_enum_flags        flags,
     /* -> Determines which databases are included in search*/
     ndr_os_db_name           search_name,
     /* -> The database name (may be wild)    */
     ndr_os_db_type_name         search_type,
     /* -> The database type (may be wild)    */
     ndr_dodb_database_id_type  search_id,
     /* -> The database id (may be wild)      */
     ndr_os_db_name           name,
     /* <- The database name                  */
     ndr_os_db_type_name         type,
     /* <- The database type */
     ndr_dodb_database_id_type  *id,
     /* <- The database id */
     UINT16                   *index);
     /* <-> Set to 0 to start else use
     previous returned value       */

//Start a new txn for this session
ndr_ret EXPORT
NdrOdStartTxn(
     ndr_od_db_handle      db_handle,
     /* -> Handle to the open DB */
     ndr_dodb_session_type   session,
     /* -> session  */
     ndr_od_txn_id          *txn_id);
     /* <- txn id   */
```

The interface includes NdrOdCloseTxn(), which closes updates for the current transaction and causes all updates since the last NdrOdStartTxn() call to be applied. Either all updates will be applied, or none will be applied. NdrOdCloseTxn() does not commit the updates, that is, they are

not written to disk. NdrOdCommit() is used to commit closed updates to disk. However, after calling NdrOdCloseTxn(), no further updates may be applied in the transaction. This function is also where all the locking and updates previously cached actually get done. Consequently, most locking and/or consistency errors are reported here (after synchronization) so that the transaction can be retried:

```
ndr_ret EXPORT
NdrOdCloseTxn(ndr_od_txn_id        txn_id);    /* -> txn_id
*/
```

The NdrOdEndTxn() function ends the current transaction and executes an implicit NdrOdCloseTxn(). No error is returned if no transaction is currently open:

```
ndr_ret EXPORT
NdrOdEndTxn(ndr_od_txn_id        txn_id);    /* -> txn id   */
```

The NdrOdCommit function commits all previously closed transactions for the session to disk:

```
ndr_ret EXPORT
NdrOdCommit(
     ndr_od_db_handle        db,        /* -> DB to commit    */
     ndr_dodb_session_type   session); /* -> session  */
```

The interface also includes the following functions:

```
//Abort current txn
ndr_ret EXPORT
NdrOdAbortTxn(ndr_od_txn_id        txn_id);    /* -> txn_id
*/

//Get info on current txn
ndr_ret EXPORT
NdrOdGetTxnInfo(
     ndr_od_txn_id        txn_id,        /* -> txn_id    */
     ndr_od_txn_info*     txn_info);     /* <- txn info */

//Lookup an object using parent Distributed Object Identifier
//(DOID; encodes location info to assist in sending distributor
//requests to the right machine; includes UOID) & sibling key
or
//using global key; the key value MUST be a contiguous
structure.
ndr_ret EXPORT
NdrOdLookupByKey(
     ndr_od_txn_id        txn_id,             /* -> txn_id   */
```

```
     ndr_dodb_access_rights_type rights_needed_on_parent,
     /* -> rights needed on parent */
     ndr_os_class          class_id,
     /* -> Class id. of superclass to match*/
     /*    Acts as filter when key contains wildcard.   */
     ndr_dodb_doid_class*    parent_doid,    /* -> Parent DOID
*/
     ndr_os_attribute      key_id,
     /* -> Type of unique key   */
     UINT16                key_length,
     /* -> Length, in bytes, of the key value */
     VOID*                 key,              /* -> Key value
*/
     ndr_dodb_doid_class* doid);
     /* <- Pointer to returned DOID of object */

//Lookup an object using DOID
//This checks the existence of the object and updates its DOID
ndr_ret EXPORT
NdrOdLookup(
     ndr_od_txn_id          txn_id,          /* -> txn_id   */
     ndr_dodb_access_rights_type rights_needed_on_parent,
     /* -> rights needed on parent */
     ndr_dodb_doid_class*    doid,           /* ->    DOID  */
     ndr_dodb_doid_class* new_doid);
     /* <- Updated DOID of object  */

//Lookup an object's parent using DOID.
ndr_ret EXPORT
NdrOdLookupParent(
     ndr_od_txn_id          txn_id,         /* -> txn_id   */
     ndr_dodb_access_rights_type rights_needed_on_parent,
     /* -> rights needed on parent */
     ndr_dodb_doid_class* doid,             /* ->    DOID  */
     ndr_dodb_doid_class* parent_doid);
     /* <- Parent DOID of object   */

//Read an object using parent DOID and sibling key or using
//global key.  It's always OK to read an object with an out of
//date parent_doid as the parent's lid is not used to get the
//reference location.  The key value MUST be a contiguous
//structure.
ndr_ret EXPORT
NdrOdReadByKey(
     ndr_od_txn_id          txn_id,         /* -> txn_id   */
     ndr_dodb_access_rights_type rights_needed_on_parent,
     /* -> rights needed on parent */
     ndr_os_class           class_id,
     /* -> Class id. of superclass to match */
     /*    and superclass structure to be returned */
     ndr_dodb_doid_class* parent_doid,  /* -> Parent DOID */
     ndr_os_attribute       key_id,  /* -> Type of unique key */
     UINT16                 key_length,
     /* -> Length, in bytes, of the key value */
     VOID*              key,         /* -> Key value   */
     UINT16                 max_length,
     /* -> Max length of data read */
     UINT16*                length,
```

```
     /* <- Final length of data read  */
     ndr_os_object*       object);
     /* => Pointer to object buffer   */

//Read an object using DOID
ndr_ret EXPORT
NdrOdRead(
     ndr_od_txn_id          txn_id,        /* -> txn_id   */
     ndr_dodb_access_rights_type rights_needed_on_parent,
     /* -> rights needed on parent */
     ndr_os_class           class_id,
     /* -> Class id. of superclass to match */
     /*    and superclass structure to be returned */
     ndr_dodb_doid_class* doid,          /* -> DOID  */
     UINT16                 max_length,
     /* -> Max length of data read */
     UINT16*                length,
     /* <- Final length of data read  */
     ndr_os_object*         object);
     /* -> Pointer to object buffer   */
```

An NdrOdReadn() function which reads multiple objects using parent DOID and wildcards behaves as if none of the updates in the transaction have been applied. Interpretation of wildcard values in the key is done by registered keying functions. NdrOdReadn() reads either up to max_objects, or up to the maximum number of objects that will fit in the max_length object buffer:

```
ndr_ret EXPORT
NdrOdReadn(
     ndr_od_txn_id          txn_id,         /* -> txn_id   */
     ndr_dodb_access_rights_type rights_needed_on_parent,
     /* -> rights needed on parent */
     ndr_os_class           class_id,
     /* -> Class id. of superclass to match
     and superclass structure to be returned */
     ndr_os_class          read_as_class,
     /* ->     Class id. target objects are to be read as   */
     ndr_dodb_doid_class* parent_doid,  /* -> Parent DOID */
     ndr_os_attribute      key_id,  /* -> Type of unique key */
     UINT16                key_length,
     /* -> Length, in bytes, of the key value */
     VOID*              key,
     /* -> Key value to match, can contain wildcard.
     NULL implies match all objects under parent containing
     the key id */
     UINT16                 max_length,
     /* -> Max length of data read */
     UINT16*                length,
     /* <- Final length of data read  */
     ndr_dodb_object_list*   object_list,
```

```
    /* -> Pointer to object buffer   */
    UINT16                    max_objects,
    /* -> Max number of objects read.  Use OD_MAX_OBJECTS to
    read max that will fit in buffer */
    ndr_dodb_context_type*  context);
    /* <> ->     set to DODB_CONTEXT_START to start a new read,

    or a previously returned context to continue a previous
    read. <-     set to DODB_CONTEXT_END if all objects read,
    or a value that can be used to continue reading at the
    next object */

#define  NDR_OD_MAX_OBJECTS   0xFFFF
```

The NdrOdLock() function explicitly adds an exclusive or shared lock to an object using the object's DOID. The lock call is called implicitly for all updates, but should be called explicitly if read locks are required. The lock is only taken when the transaction is initially executed. It is not executed when the update is merged. The lock is applied at the end of a transaction. If it fails the transaction is aborted and should be re-tried by the caller. One embodiment does not utilize locks to control concurrency but instead relies on retries and clash handling:

```
ndr_ret EXPORT
NdrOdLock(
    ndr_od_txn_id         txn_id,  /* -> txn_id   */
    ndr_dodb_doid_class* doid,     /* -> Objects's DOID */
    BOOLEAN              is_exclusive);
    /* -> TRUE => take exclusive lock   */
```

The interface also includes:

```
//Add agent defined lock to object
ndr_ret EXPORT
NdrOdAddAgentLock(
    ndr_od_txn_id         txn_id,  /* -> txn_id   */
    ndr_dodb_doid_class* doid,     /* -> Objects's DOID */
    ndr_dodb_lock_type        lock_type,
    /* -> Type of lock           */
    ndr_dodb_lock_flags_type lock_flags,
    /* -> Flags that allow multiple locks to be taken
    in single call.  Each bit corresponds to a separate
    lock, e.g. used for read/write flags on file open   */
    ndr_dodb_lock_deny_flags_type deny_flags);
    /* -> Bits set that correspond to lock_flags bits
    causes the corresponding lock to be denied  */
```

```
//Remove agent defined lock
ndr_ret EXPORT
NdrOdRemoveAgentLock(
     ndr_od_txn_id          txn_id,  /* -> txn_id   */
     ndr_dodb_doid_class* doid,      /* -> Objects's DOID */
     ndr_dodb_lock_type         lock_type);
     /* -> Type of lock                 */
```

The following four calls are used to append various types of updates onto an open transaction. Any of them may return NDR_OK indicating success, NDR_CD_EXCEEDED_TXN_LIMITS indicating that transaction limits have been exceeded, or some other error indicator. In the case of exceeded transaction limits the transaction state will not have been changed and the failed call will have had no effect. The caller is expected to commit or abort the transaction as appropriate. In all other error cases the transaction is automatically aborted before returning the error to the caller:

```
//Modify a single attribute in a previously read object
//The object distributor caches the modifications and only
//applies them at close txn time
ndr_ret EXPORT
NdrOdModifyAttribute(
     ndr_od_txn_id          txn_id,         /* -> txn_id   */
     ndr_dodb_access_rights_type rights_needed_on_parent,
     /* -> rights needed on parent */
     ndr_dodb_doid_class* doid,
     /* ->    DOID of previous read version of object.
     Used to verify object has not been modified by another
     user since previously read */
     ndr_os_attribute      attribute_id,
     /* -> Identifies attribute to be modified    */
     VOID*                value);     /* -> New attribute value */

//Add a new object
//The DOID attribute does not need to be filled in by the
caller.
//The DOID will be set up before writing the object to the
//database.
ndr_ret EXPORT
NdrOdAdd(
     ndr_od_txn_id          txn_id,         /* -> txn_id   */
     ndr_dodb_access_rights_type rights_needed_on_parent,
     /* -> rights needed on parent */
     ndr_dodb_doid_class* parent_doid,  /* -> Parent DOID */
     ndr_os_class           class_id,
     /* ->   Class id of object */
     ndr_os_object*         object);
```

```
     /* -> Pointer to agent object      */

//Remove an object using DOID
ndr_ret EXPORT
NdrOdRemove(
     ndr_od_txn_id          txn_id,        /* -> txn_id   */
     ndr_dodb_access_rights_type rights_needed_on_parent,
     /* -> rights needed on parent */
     ndr_dodb_doid_class* doid);           /* -> DOID  */

//Move an object using DOID
ndr_ret EXPORT
NdrOdMove(
     ndr_od_txn_id          txn_id,        /* -> txn_id   */
     ndr_dodb_access_rights_type rights_needed_on_parent,
     /* -> rights needed on parent */
     ndr_dodb_doid_class* doid,            /* -> DOID  */
     ndr_dodb_doid_class* target_parent_doid);
     /* -> Target parent DOID   */

//Set a marker in an open transaction.  The state of the
//transaction at the time the marker is set can be reverted
//to at any time before the transaction is closed by
//calling NdrOdRevertToMarker().
//Only the last marker in a transaction is significant.
//This call may return NDR_CD_EXCEEDED_TXN_LIMITS which
//should be treated as for the update appending calls above
ndr_ret   EXPORT
NdrOdSetMarker(ndr_od_txn_id       txn_id);  /* -> txn_id   */

//Revert a txn's state to the last previously marked state
ndr_ret   EXPORT
NdrOdRevertToMarker(ndr_od_txn_id  txn_id);  /* -> txn_id   */

//Add a <user-id, rights-mask> pair to an object's
//access rights, overwriting any previous rights-mask for
//that user
ndr_ret EXPORT
NdrOdAddAccessRight(
     ndr_od_txn_id              txn_id,    /* -> txn_id   */
     ndr_dodb_doid_class*       doid,      /* -> Object DOID */
     ndr_dodb_auth_id_type      user,
     /* -> User to whom rights are to be granted  */
     ndr_dodb_access_rights_type rights);
     /* -> Rights to be granted to that user  */

//Remove any <user-id, rights-mask> pair from an object's
//access rights for a given user-id
ndr_ret EXPORT
NdrOdRemoveAccessRight(
     ndr_od_txn_id              txn_id,    /* -> txn_id   */
     ndr_dodb_doid_class*       doid,      /* -> Object DOID */
     ndr_dodb_auth_id_type      user);
     /* -> User whose rights are to be revoked  */

//Get the array of all <user-id, rights-mask> pairs for an
object
ndr_ret EXPORT
```

```
NdrOdGetAccessRights(
     ndr_od_txn_id         txn_id,       /* -> txn_id    */
     ndr_dodb_doid_class* doid,          /* -> Object DOID */
     UINT16*               acl_count,
     /* <- Number of ACL entries for that object */
     ndr_dodb_acl_element_type* acl);
     /* <- Rights information for that object   */

//Get the effective access rights for the current session
//for an object
ndr_ret EXPORT
NdrOdGetEffectiveAccessRight(
     ndr_od_txn_id              txn_id,   /* -> txn_id    */
     ndr_dodb_doid_class*       doid,     /* -> Object DOID */
     ndr_dodb_access_rights_type* rights);
     /* <- Effective rights for the current session  */

//Convert UOID to DOID
ndr_ret EXPORT
NdrOdConvertUoid2Doid(
     ndr_os_class              class_id,
     /* -> Class id. of object  */
     ndr_dodb_uoid_type*       uoid,      /* -> UOID  */
     ndr_dodb_doid_class*      doid);     /* <- Updated DOID */

//Convert UOID to DOID
ndr_ret EXPORT
NdrOdConvertUoid2LocalDoid(
     ndr_os_class              class_id,
     /* -> Class id. of object  */
     ndr_dodb_lid_type         location,
     /* -> Location on which object exists */
     ndr_dodb_uoid_type*       uoid,      /* -> UOID  */
     ndr_dodb_doid_class*      doid);     /* <- Updated DOID */
```

The object processor 86 provides a local hierarchical object-oriented database for objects whose syntax is defined in the object schema 84. In one embodiment, the object processor 86 is built as a layered structure providing functionality according to an interface in the structure which is described below. The embodiment also includes a module for object attribute semantics processing, a set of global secondary indexes, a hierarchy manager, a B-tree manager, a record manager, and a page manager. Suitable modules and managers are readily obtained or constructed by those familiar with database internals. A brief description of the various components follows.

The page manager provides functionality according to a logical file interface of free-form fixed length pages addressed by logical page number. Rollback and commit at this level provide anti-crash recovery.

The record manager provides for the packing of variable length keyed records into fixed length pages.

The B-tree manager uses the facilities of the record and page managers to provide general B-trees supporting variable length records and variable length keys.

The hierarchy manager imposes a hierarchical structure on records by use of structured B-tree keys and a global UOID->full name index.

The secondary index manager provides generalized global indexing capabilities to records.

The attribute manager interprets the schema 84 in order to raise the interface of the object processor 86 from a record-level to an object-level interface.

The interface module of the object processor 86 uses lower level interfaces to provide functionality according to the following interface:

op_init	Initializes object processor
op_shutdown	Shuts down object processor
op_add_database	Creates a new volume
op_mount_database	Mounts a specified volume for use
op_dismount_database	Dismounts the specified volume
op_remove_database	Removes a specified volume (permanently)
op_read	Read an object by UOID
op_readn	Read one or more objects with wildcards
op_execute_update_list	Apply one or more updates
op_commit	Commit updates to a specified volume
op_rollback	Rollback to the last committed state
op_free_inversion_list	Free up an inversion list returned from update execution
op_clear_stats	Clear object processor statistics

op_dump_stats Dump statistics to the log

Due to higher level requirements of trigger functions in a set of trigger function registrations 94, in one embodiment it is necessary to have the old values of modified attributes available on a selective basis. This is done by means of a 'preservation list' produced by op_execute_updates(). The preservation list contains an update list specifying old attribute values for all executed updates that require it (as determined by a callback function), together with pointers to the original causative updates. These updates may not actually be present in the input update list, as in the case of an object removal that generates removes for any descendant objects it may have. Preservation lists reside in object processor 86 memory and must thus be freed up by the caller as soon as they are no longer needed.

The transaction logger 88 provides a generic transaction log subsystem. The logs maintained by the logger 88 provide keyed access to transaction updates keyed according to location identifier and processor transaction identifier (PTID). In one embodiment, a non-write-through cache is used to batch uncommitted transaction updates.

The transaction logger 88 is used by the consistency processor 76 to support fast recovery after a crash. Recovery causes the target database to be updated with any transactions that were committed to the log by the logger 88 but were not written to the target database. The log file header contains a "shutdown OK" flag which is used on startup to determine if recovery is required for the location.

The transaction logger 88 is also used by the consistency processor 76 to support fast synchronization. The update log created by the logger 88 is used to replay the updates from one location to a second location using minimal disk and network 10 transfers.

The file distributor 90 distributes file contents to appropriate locations in the network 10. A file processor 92 supports each file distributor 90 by carrying out requested read, write, lock, or other operations locally.

The file distributor 90 hides from agents the complexities caused by the distributed nature of files. To the extent possible, the interface portion of the file distributor 90 resembles file system interfaces that are familiar in the art. An open file is denoted by a numeric fork_id and functions are provided to read, write, open, and otherwise manipulate and manage files and their contents.

However, a class in the schema 84 can be given a REPLICATED_FILE property. Whenever an object of such a class is created in the database, a distributed file is created by the file distributor 90 and file processor 92 to hold the file contents associated with that object. For instance, the Hierarchy Agent might create such an object to denote a leaf node in the directory hierarchy. In short, in one embodiment the file distributor 90 neither has nor needs an explicit externally called mechanism for creating files.

Moreover, the distributed file is deleted from storage when the corresponding object is deleted from the database. The locations at which the file is stored are precisely those at which the object exists. When a file with more than one

replica 56 is modified and closed, the file distributors 90 and file processors 92 at the various locations holding the replicas 56 ensure that all replicas 56 of the file receive the new contents. It is not necessary for the agent to expressly manage any aspect of file content distribution.

A distributed file is identified by the UOID of the corresponding object; no built-in hierarchical naming scheme is used. A transaction identifier is also required when opening a file, to identify the session for which the file is to be opened. In one embodiment, the file distributor 90 and file processor 92 provide functionality according to the following interface:

```
//An ndr_fd_fork_id is the Id by which an FD open fork is known
typedef SINT16 ndr_fd_fork_id;
#define  NDR_FD_NOT_A_FORK_ID (-1)
//An ndr_fd_open_mode is a bit-mask which specifies whether a
//fork is open for reading and/or writing
typedef UINT16 ndr_fd_open_mode;
#define   NDR_FD_OPEN_READ_MODE          0x0001
#define   NDR_FD_OPEN_WRITE_MODE         0x0002
#define   NDR_FD_OPEN_EXCL_MODE          0x0004
#define   NDR_FD_OPEN_EXTERNAL_MODES     0x0007
//The remaining open modes are private to the replica managers
#define   NDR_FD_OPEN_SYNC_MODE          0x0008
#define   NDR_FD_OPEN_CLOSE_ON_EOF_MODE  0x0010
#define   NDR_FD_OPEN_READ_NOW           0x0020
```

In one alternative embodiment, opening a file with an NdrFdOpenFile() function returns pointers to two functions together with a separate fork_id for use with these two functions only. These pointers are of the type ndr_fd_io_function, and may be used as alternatives to NdrFdReadFile() and NdrFdWriteFile() when accessing that open file only. The functions should be at least as efficient as NdrFdReadFile() and NdrFdWriteFile() and will be significantly faster when the file access is to a local location. Their use does require that the caller maintain a mapping from the open

fork id onto these function pointers. For this reason, NdrFdReadFile() and NdrFdWriteFile() should always be available for all open files in this alternative embodiment:

```
typedef  ndr_ret EXPORT (*ndr_fd_io_function)(
     ndr_fd_fork_id          fork_id,       /* -> Id of open fork
*/
     UINT32                  offset,
     /* -> Offset at which to start reading */
     UINT16*                 length,
     /* <->   desired length on entry, actual length on
     exit.  These will only differ if an error
     is encountered (such as end of file)       */
     UINT8*                  data,
     /* <->   Data read or written */
     ndr_od_txn_id           txn_id);       /* -> txn_id    */
```

A "clash" occurs during synchronization when two desired changes to the database are inconsistent. Clashes arise from "independent" updates, namely, updates performed on separate replicas 56 while the computers holding the replicas 56 were disconnected. Thus, clashes always take place between a pair of "clashing updates" which together define a "clash condition." A "repairing update" is an update that removes a clash condition caused by a clashing update.

A "transient clash" is a clash that is not present in the final states of the two replicas 56 being merged. Transient clashes only arise when log-based or hybrid merging is used. For instance, suppose two users each create a file of a given name at two locations 36, 38 while those locations are disconnected. The user at the first location 36 then deletes (or renames or moves) the file in question before reconnection such that it no longer clashes with anything on the second location 38. On merging the replicas 56 of the two locations 36, 38, the original add update for the file from the first location 36 will clash with the replica 56 of the second location 38, yet the final result of applying the update stream

from the first location 36 to the replica 56 on the second location 38 is a state that is compatible with that replica 56.

By contrast, "persistent clashes" create inconsistencies that are present in the final states of two replicas 56. A clash whose type is unknown is a "potential clash."

A "file contents clash" occurs when a file's contents have been independently modified on two computers 28, or when a file has been removed from one replica 56 and the file's contents have been independently modified on another replica 56.

An "incompatible manipulation clash" occurs when an object's attributes have been independently modified, when an object has been removed in one replica 56 and the object's attributes have been modified in another replica 56, when an object has been removed in one replica 56 and moved in the hierarchy in another replica 56, when a parent object such as a file directory has been removed in one replica 56 and has been given a child object in another replica 56, or when an object has been independently moved in different ways. Thus, although clashes are discussed here in connection with files and the file distributor 90, clashes are not limited to updates involving files.

A "unique key clash" occurs when two different objects are given the same key and both objects reside in a portion of the database in which that key should be unique. In a database representing a file system hierarchy, for instance, operations that add, move, or modify files or directories may create a file or directory in one replica 56 that clashes on reconnection with a different but identically-named file or directory in another replica 56.

A "permission clash" occurs when a change in file access or modification permissions that is made to a central server replica 56 would prohibit an independent update made to a mobile or client computer replica 56 from being applied to the server replica 56. A permission clash is an example of an "external clash," namely, a clash detected by reference to a structure external to the database. Permission clashes and other external clashes may be detected by trigger functions.

A "grouped attribute" is a database object attribute that is associated with other database object attributes such that changing the value of any attribute in a group creates a clash with the other attributes in the group. For instance, filename and rename-inhibit attributes are preferably grouped together, while filename and file-access-date attributes are preferably not grouped together. Without attribute grouping, a change to any attribute of an object is assumed to clash with a change to any other attribute of the object or another change to the same attribute.

"Eliminating a clash" means identifying the basis for the clash and eliminating it. "Recovering from a clash" means identifying the basis for the clash and either eliminating that basis or presenting alternative resolutions of the clash to a user to choose from. "Regressing an update" means undoing the update on at least one replica 56. Creating a "recovery item" means creating a duplicate object in a shadow database and then remapping uses of the recovery item's key so that subsequent updates are performed on the recovery item instead of the original object. If the database represents a file system hierarchy, recovery items may be gathered in a "single

directory hierarchy" or "recovery directory" that contains a directory at the root of the volume, recovered items, and copies of any directories necessary to connect the recovered items properly with the root.

A clash handler function of one of the types below can be registered with the file distributor 90 for a database type to be called whenever the file distributor 90 detects a clash caused by disconnected modification or removal of a file's contents. The parameters are those of a regular clash handler plus the object DOID with NDR_OS_CLASS_FLAG_HAS_PARTIALLY_REPLICATED_FILE property (the file object defined by the object schema 84) and possibly a duplicated object return:

```
//Call back to a husk in respect of clashes detected at the
//database level
typedef  ndr_ret EXPORT (*ndr_fd_object_clash_fn)(
     ndr_od_db_handle            db,         /* -> Database */
     ndr_dodb_session_type       session,
     /* -> session to use in od_start_txn */
     ndr_od_clash_info*          info,
     /* -> Information on clash */
     ndr_dodb_doid_class*        old_doid,
     /* -> DOID of file with clashing contents      */
     ndr_dodb_doid_class*        new_doid);
     /* <- Doid of duplicated file                  */

//Call back to the husk in respect of clashes detected at the
//filesystem level
// (via pre trigger functions)
typedef  ndr_ret EXPORT (*ndr_fd_filesys_clash_fn)(
     ndr_od_db_handle            db,         /* -> Database */
     ndr_dodb_session_type       session,
     /* -> session to use in od_start_txn */
     ndr_od_clash_info*          info,
     /* -> Information on clash */
     ndr_dodb_doid_class*        doid);
     /* -> DOID of file with clashing contents      */
```

A parameter block such as the following is passed to clash handling functions to provide them with information about the clash:

```
typedef struct
```

```
{
     ndr_dodb_ptid_type*      ptid;
     /* -> PTID of clashing txn                          */
     ndr_od_clash_type        clash_type;
     /* -> Clash type                                    */
     ndr_os_class             class_id;
     /* -> Class id of object causing the clash          */
     ndr_os_attribute         attr_id;
     /* -> Attr id of object causing the clash           */
     ndr_dodb_update_list*    update_list;
     /* -> Update list of transaction                    */
     ndr_dodb_update*         update;
     /* -> Update causing clash (always a pointer
       into 'update_list'                                */
     BOOLEAN                  is_higher_priority;
     /* -> Relative priority of location
          to which update is being applied.
          TRUE=> Applying to location with higher
          priority (e.g. to location set with
          central location)                              */
     void*                         agent_merge_info;
     /* -> Value which is reserved for (arbitrary)
       use by agent clash handlers.  It is
       guaranteed to be set to NULL on the
       first clash of a merge, and preserved
       for all subsequent clashes within that
       merge                                             */
} ndr_od_clash_info;
```

A close handler function of type ndr_fd_close_fn can be registered with the file distributor 90 for a database type to be called whenever the file distributor 90 closes a modified local copy of the file contents, passing the new length and modification date/time and user identifier:

```
typedef  ndr_ret EXPORT (*ndr_fd_close_fn)(
     ndr_od_db_handle         db,        /* -> Database */
     ndr_dodb_session_type    session,
     /* -> session to use in od_start_txn */
     ndr_os_class             class_id,
     /* -> Class ID of file */
     ndr_dodb_uoid_type*      uoid,       /* -> UOID  */
     UINT32                   length,
     /* -> length of closed file */
     UINT16                   time,
     /* -> modification time */
     UINT16                   date,
     /* -> modification date */
     UINT32                   updator);
     /* -> modification user */
```

A creation handler function of type ndr_fd_creation_fn can be registered with the file distributor 90 for a database type

to be called whenever the file distributor 90 creates a local copy of the file contents. This allows the replica manager 46 on a central server computer 28 to update the master copy of the file to reflect the attributes of the file created while disconnected:

```
typedef  ndr_ret EXPORT (*ndr_fd_creation_fn)(
     ndr_od_txn_id               txn_id,      /* -> txn_id    */
     ndr_os_class                class_id,
     /* -> Class ID of file */
     ndr_dodb_uoid_type*     uoid);        /* -> UOID of file */
```

The file distributor 90 embodiment also provides the following:

```
//Return aggregated information about all volumes
ndr_ret EXPORT
NdrFdVolumeInfo(
     ndr_od_txn_id           txn_id,          /* -> txn_id    */
     UINT32*                 cluster_size,
     /* <- Number of bytes per cluster  */
     UINT16*                 total_clusters,
     /* <- Total number of clusters     */
     UINT16*                 free_clusters);
     /* <- Number of free clusters      */

//Add a file
ndr_ret EXPORT
NdrFdAddFile(
     ndr_od_txn_id           txn_id,      /* -> txn_id    */
     ndr_dodb_doid_class* doid,
     /* -> Uoid of file created */
     UINT32                  length);
     /* -> Length of existing file (0 when new) */

//Remove a file
ndr_ret EXPORT
NdrFdRemoveFile(
     ndr_od_txn_id          txn_id,      /* -> txn_id    */
     ndr_dodb_uoid_type* uoid);
     /* -> Uoid of file removed */

//Open a file for reading or writing by a task
ndr_ret EXPORT
NdrFdOpenFile(
     ndr_od_txn_id          txn_id,   /* -> txn_id    */
     ndr_os_class           class_id,
     /* -> Class ID of file to open */
     ndr_dodb_uoid_type  uoid,
     /* -> Uoid of file to open */
     ndr_fd_open_mode    open_mode,
     /* -> Open for read and/or write? */
     ndr_fd_fork_id*     fork_id,
```

```
        /* <- FD Fork Id of open file */
        BOOLEAN                 is_create,
        /* ->      TRUE if open as part of create      */
        ndr_fd_io_function* read_function,
        /* <- Function to be used for READ operations */
        ndr_fd_io_function* write_function,
        /* <- Function to be used for WRITE operations */
        ndr_fd_fork_id*       io_fork_id,
        /* <- FD Fork Id used with above two functions (only) */
        UINT16*                 num_forks_remaining);
        /* <- Number of forks remaining to be opened
        on same machine      */

    //Read from a file
    ndr_ret EXPORT
    NdrFdReadFile(
        ndr_od_txn_id          txn_id,  /* -> txn_id    */
        ndr_fd_fork_id         fork_id, /* -> Id of open fork */
        UINT32                 offset,
        /* -> Offset at which to start reading */
        UINT16                 req_length,
        /* -> Number of bytes requested to read */
        UINT8*                 data,    /* <- Data read */
        UINT16*                act_length);
        /* <- Actual number of bytes read */

    //Write to a file
    ndr_ret EXPORT
    NdrFdWriteFile(
        ndr_od_txn_id          txn_id,  /* -> txn_id    */
        ndr_fd_fork_id         fork_id, /* -> Id of open fork */
        UINT32                 offset,
        /* -> Offset at which to start writing */
        UINT16                 req_length,
        /* -> Number of bytes requested to write */
        UINT8*                 data);   /* -> Data to be written */

    //Get the current length of an open file
    ndr_ret EXPORT
    NdrFdGetOpenFileLength(
        ndr_od_txn_id          txn_id,  /* -> txn_id    */
        ndr_fd_fork_id         fork_id, /* -> Id of open fork */
        UINT32*                length);
        /* <- Length of that open file */

    //Lock or Unlock a range of bytes in an open file
    ndr_ret EXPORT
    NdrFdClearPhysicalRecord( or NdrFdLockPhysicalRecord(
        ndr_od_txn_id          txn_id,  /* -> txn_id    */
        ndr_fd_fork_id         fork_id, /* -> Id of open fork */
        UINT32                 offset,  /* -> Offset for lock */
        UINT32                 req_length);
        /* -> Number of bytes requested to lock */

    //Ensure a file's contents are on disk
    ndr_ret EXPORT
    NdrFdCommitFile(
        ndr_od_txn_id          txn_id,       /* -> txn_id    */
```

```
    ndr_fd_fork_id          fork_id);       /* -> Id of open fork
*/

//Close a file, having completed reading and writing
ndr_ret EXPORT
NdrFdCloseFile(
    ndr_od_txn_id           txn_id,         /* -> txn_id    */
    ndr_fd_fork_id          fork_id);       /* -> Id of open fork
*/

//Given a UOID to a file or directory return its name
//in the specified namespace, along with its parent's UOID
ndr_ret   EXPORT
NdrFdGetFilename(
    ndr_od_db_handle            db,
    /* ->     handle to current database */
    ndr_dodb_uoid_type*         file_or_dir_id,
    /* ->     Uoid of object whose name is wanted */
    ndr_os_attr_property        namespace,
    /* ->     Namespace (e.g. DOS) of name wanted */
    void*                             name_buffer,
    /* <-     Buffer to receive name */
    UINT16*                           name_size,
    /* ->     Size of provided buffer */
    ndr_dodb_uoid_type*         parent_dir_id);
    /* <-     Parent UOID of object (NULL at root) */

//Callback functions to be used with
//NdrFdRegisterChangedIdCallback
typedef ndr_ret     EXPORT
(*NdrFdChangedIdCallback)(
    ndr_od_db_handle      db,           /* -> Database Id */
    ndr_os_class          class_id,
    /* -> Class ID of file or dir */
    ndr_dodb_uoid_type    *uoid,        /* -> Uoid of file or dir
*/
    UINT32                new_id);
    /* -> New Id allocated by underlying file system */
```

A NdrFdRegisterChangedIdCallback() function provides registration of a callback function to be called when a change to a file or directory's unique identifier is made. On a NetWare 4.x server this normally happens only when the file or directory is created by an internal file distributor 90 trigger function. However the identifier will be needed by agents for tasks such as directory enumeration. Because trigger functions cannot directly modify replicated objects, a record of the identifier change is queued within the file distributor 90 and the callback is made asynchronously:

```
ndr_ret EXPORT
NdrFdRegisterChangedIdCallback(
     ndr_os_db_type_handle   db_type, /* -> Database type */
     NdrFdChangedIdCallback  fn); /* -> Callback function */
```

The interface also provides the following:

```
//Register clash handlers for contents clashes for files held
in
//a database of the given type.
ndr_ret EXPORT
NdrFdRegisterClashHandlers(
     ndr_os_db_type_handle    db_type,  // -> Database type
     ndr_os_class             class_id,
     // -> Class ID of contents 'container' eg file
     ndr_fd_object_clash_fn   object_clash_fn,
     // -> Clash handler for dealing with conflicts
     // -> between objects (e.g. contents modification
     //      and removal)
     ndr_fd_filesys_clash_fn  filesys_clash_fn,
     // -> Clash handler for conflicts that arise
     //     through some characteristic of the file
     //     system (e.g. access rights on delete)
     ndr_fd_filesys_clash_fn  filesys_clash_fn1);

//Register a trigger-like routine to be called when a local
//replica of a file is modified. The routine takes the length
//and modification date/time of the local replica of the file.
ndr_ret EXPORT
NdrFdRegisterCloseHandler(
     ndr_os_db_type_handle    db_type, // -> Database type
     ndr_os_class             class_id,
     /* -> Class ID of file */
     ndr_fd_close_fn          close_fn);
     /* -> Clash handler to call */

//Register a trigger-like routine to be called when a local
//replica of a file is has been created. This allows the
//replica manager on a central server to update the
//server's master copy of the file to reflect the attributes
//of the file created during the disconnection.
ndr_ret EXPORT
NdrFdRegisterCreationHandler(
     ndr_os_db_type_handle    db_type, /* -> Database type */
     ndr_os_class             class_id,
     /* -> Class ID of file */
     ndr_fd_creation_fn       creation_fn);
     /* -> Creation handler to call */

//De-register a clash or close or creation handler for
//contents clashes for files held in a database of the given
type
ndr_ret EXPORT
NdrFdDeRegisterClashHandler( or CloseHandler( or
CreationHandler(
     ndr_os_db_type_handle    db_type, // -> Database type
     ndr_os_class             class_id); // -> Class ID of file
```

```
//Synchronize all the files to and from this client for the
//passed database.  Return control when the files are up to
date.
ndr_ret EXPORT
NdrFdSynchronizeFiles(ndr_od_db_handle  db);

//Called from pre trigger functions to check whether
//or not the current connection has sufficient
//per-user-rights to perform a particular operation
//on a particular file system object.
ndr_ret
NdrFdCheckRights(
     ndr_dodb_uoid_type*          file_uoid,
     // uoid of object requiring rights to operation
     ndr_od_db_handle             db,
     // database raising the pre trigger
     UINT16                       operation);
     // bits representing operation

//Note that a file has been locally modified, setting
//modification info and triggering propagation onto other
//replicas.
ndr_ret EXPORT
NdrFdNoteFileModified(
     ndr_od_txn_id           txn_id,     /* -> txn_id   */
     ndr_dodb_doid_class*    file_doid);
```

The trigger function registrations 94 identify trigger functions that are provided by agents and registered with the object distributor 82. A registered trigger function is called on each event when the associated event occurs. Suitable events include object modification events such as the addition, removal, movement, or modification of an object. Because the trigger functions are called on each location, they can be used to handle mechanisms such as file replication, where the file contents are not stored within the target database, while ensuring that the existence, content, and location of the file tracks the modifications to the target database. All objects must have been locked, either implicitly or via NdrOdLock(), in the triggering transaction before the corresponding trigger function is called, and other objects may only be modified if the trigger function is being called for the first time at the location in question.

In an alternative embodiment, the replica manager 46 comprises a NetWare Loadable Module ("NLM") and an NWAdmin snap-in module. The NLM uses hooks in the NetWare file system 48 to intercept updates to the local NetWare storage 54, and uses standard NetWare file system Application Programmer's Interface ("API") calls to update the storage 54 when synchronizing. The architecture is symmetric, with the same code running on all computers 28.

The NLM has three major internal subsystems. An environment subsystem provides portability by separating the other two internal subsystems from the operating system environment such as the Windows NT or UNIX environment. The environment subsystem provides execution, debugging, scheduling, thread, and memory management services. A Distributed NetWare ("DNW") subsystem implements NetWare semantics by intercepting NetWare file system calls and calls from a DNW API and making corresponding requests of a dispatch layer discussed below. A distributed responder subsystem implements the replica manager 46 to provide a distributed disconnectable object database which supports replication, transaction synchronization, and schema-definable objects, including file objects, as described herein.

An application layer contains application programs and the NWAdmin snap-in. These programs interface with the replica manager 46 either by calling an API or by attempting to access the storage device 54 and being intercepted. An intercept layer in the replica manager 46 intercepts and routes external requests for file system updates that target a replica 56. A

dispatch later receives the routed requests and dispatches them to an appropriate agent 44.

The agents 44, which have very little knowledge of the distributed nature of the database, invoke the consistency distributor 74, location distributor 78, object distributor 82, and/or file distributor 90. For example, a directory create would result in an object distributor 82 call to NdrOdAdd() to add a new object of type directory.

In contrast to the agents 44, the distributors 74,78,82, and 90 have little semantic knowledge of the data but know how it is distributed. The object distributor 82 uses the location distributor 78 to control multi-location operations such as replication and synchronization. The consistency distributor 74 manages transaction semantics, such as when it buffers updates made after a call to NdrOdStartTxn() and applies them atomically when NdrOdEndTxn() is called. The file distributor 90 manages the replication of file contents.

The processors 76, 86, 88, and 92 process requests for the local location 40. The consistency processor 76 handles transaction semantics and synchronization, and uses the transaction logger 88 to log updates to the database. The logged updates are used to synchronize other locations 40 and to provide recovery in the event of a clash or a crash. The logger 88 maintains a compressed transaction log. The log is "compressed," for example, in that multiple updates to the "last time modified" attribute of a file object will be represented by a single update. The logger 88 maintains a short sequential on-disk log of recent transactions; the

longer-term log is held in the object database as update log entry objects.

The object processor 86 implements a local object store and supports the following access methods: hierarchical (e.g., add file object under directory object); global indexed (e.g., read any object using its UOID); and local indexed (e.g., read files and directories within a directory in name order). The object processor 86 uses a variant of a B*-tree. The object processor 86 uses a page table to support atomic commitment of transactional updates, providing rollback and protection against crashes of the computer 40.

A file system layer in the file system interface 48 provides a flat file system interface built on the local host file system. It re-maps the flat file system calls to the corresponding files in the hierarchical NetWare volumes to support the current NetWare file system.

With reference to Figures 1 through 4 and particular focus on Figure 4, methods of the present invention for managing a transaction log are illustrated. One method of the present invention compresses the transaction log by utilizing an update identifying step 100 to identify a redundant update and a subsequent update removing step 102 to remove that update. These steps, as well as other steps shown in Figure 4, may be repeated in additional iterations; the bottom of the flowchart leads back to the top as well as to other management operations such as transaction synchronization steps or clash handling steps.

A variety of redundant updates are identified by replica managers 46 or other systems which operate according to the

methods of the present invention. For instance, a file or directory may be renamed twice, rendering the first rename redundant. Likewise, a file may be modified twice, rendering the first update to the modification date redundant. Scripts or other mechanisms may also repeat operations to no further effect, such as deleting a file twice without recreating it in between the deletes or moving a file and then immediately returning it to its original location. These and similar redundant update sequences are identified during the step 100.

More complex but nonetheless redundant sequences can also be analyzed during the step 100. For instance, use of the location state processor 80 may identify an update in the transaction log that specifies an update location on a computer 40 other than the computer 40 which holds the log presently being managed. The log can then be compressed during the step 102 by removing that update.

In other situations, further steps are employed to identify redundant updates. For instance, a transaction identifying step 104 determines the most recent successfully merged transaction that updates a selected object. Transaction boundaries may be identified by checkpoints inserted in the log during transaction synchronization or by version control operations. Boundaries may be determined using the object processor 86 as described below in connection with certain three-level structures. Every transaction checkpoint is located at the boundary of a transaction, as defined by the three-level structures or other means, that is found in the log prior to compression. However, not every such boundary will be available as a checkpoint because compression may remove some

boundaries. Checkpoints may act as constraints on compression by delimiting incompressible sequences of updates.

An update of the selected object that antedates the transaction is next identified during the step 100. Finally, the update is removed during the step 102. This presumes that committed updates will not be needed again. In an alternative embodiment, committed updates are retained to permit recovery from log corruption or to permit the reproduction of earlier versions of objects for other reasons.

To rule out certain updates as candidates for removal, an incompressible sequence identifying step 106 is performed in one embodiment. An otherwise removable update will not be removed from an incompressible sequence. A sequence which spans a transaction boundary is incompressible if removing an update from the sequence would require the propagation to replicas 56 of changes to a committed transaction. A sequence which semantically relies on a temporary item to swap two other items cannot be compressed; an example would be the sequence:

```
rename A X
rename B A
rename X B
```

To facilitate the identifying and/or removing steps 100, 102, the replica manager 46 may reposition an update in the sequence of updates in the transaction log during a repositioning step 108. One or more repositioned updates may then be consolidated with an adjacent update during a consolidating step 110. Consolidation replaces two or more updates with a single equivalent update.

For instance, the sequence of updates "add A; add B; add C; rename A Z" may be managed by the repositioning step 108 to

produce "add A; rename A Z; add B; add C" and subsequently managed by the consolidating step 110 to produce the sequence "add Z; add B; add C." Of course, other semantically equivalent sequences may also be produced according to the invention, such as the sequence "add B; add C; add Z."

In short, redundant updates are identified by examining the operations performed by the updates and the status of the replicas 56. Log compression is based on update semantics, unlike data compression methods such as run-length-encoding which are based only on data values.

During a creating step 112, a hierarchical log database is made. The log database represents at least a portion of the transaction log using objects which correspond to the updates and transactions in the specified portion of the transaction log. The log database is preferably efficiently cached and swapped to disk as needed.

In one embodiment, the log database is a hierarchical database maintained by the object processor 86. Transactions are represented as a three-level structure. A top-level transaction sequence object contains the PTID associated with the transaction that is described by the object's descendant objects. This PTID is a global key, with its sibling key being the log append order for the transaction in question.

An update container object, which is a child of the transaction sequence object, serves as the parent of the transaction's log database update objects. It is separated from the transaction sequence object in order to allow the updates in a transaction to migrate into another (by PTID) transaction during the repositioning step 108.

One or more update objects, which are the children of the update container object, represent individual updates to the target database during the transaction. Update objects are separated from one another to aid analysis during the update identifying step 100 and to reduce the effect of possible object size limitations. Update objects are ordered by a sibling key which preserves their order within the update container.

In an alternative embodiment, the three-level structure is replaced by a physically flattened, more efficient structure which nonetheless provides logical separation of transaction objects and update objects by use of keys. The namespace of the keys is partitioned to permit interleaving of update and transaction objects under a common parent. The intermediate level, represented in the previously described embodiment by update container objects, is not present.

In one embodiment, update compressibility relationships are represented using a "previous update" field in each update object. This field contains the UOID of the update object which represents the most recent previous update in the log for the object in question. A NULL UOID, present only in objects representing a create-performing update, indicates that no such previous update is present.

Optional synchronization checkpoints provide a way to group transactions. Checkpoints are represented using multi-valued attributes in transaction sequence objects or one or more checkpoint objects. In one embodiment, one checkpoint object per checkpoint is present; in an alternative embodiment one checkpoint object represents all synchronization

checkpoints. Each checkpoint attribute or checkpoint object contains the location identifier value(s) of the location(s) 40 to which the synchronization checkpoint pertains. In the case where the log is on a client 20, this will be the corresponding server 16. If no values are present, no synchronization has yet been done.

The portion of the transaction log represented by the log database may be the entire log, or it may be a smaller portion that only includes recent transactions. In the latter case, the remainder of the transaction log is represented by a compressed linearly accessed log stored on the device 54. In embodiments that do not include a log database, the entire transaction log is represented by a linearly accessed log stored on the device 54.

During one or more iterations of an inserting step 114 objects are inserted into the log database to represent updates, transactions, or synchronization checkpoints. Updates are represented as individual objects and determination of necessary management steps is often made at the update level.

However, the desired transactional treatment of updates requires that updates in a given transaction are always committed to the replica 56 together. Thus, in inserting an update as described herein, the replica manager 46 actually inserts a transaction containing that update. Likewise, in consolidating two updates from separate transactions, the replica manager 46 consolidates the transactions. And in moving an update, the replica manager 46 moves an entire transaction to make two transactions needing consolidation become adjacent to each other.

One method of the present invention appends a transaction to the log by inserting a new three-level transaction structure into the log database. During an accessing step 116 an update history structure is created or modified when the transaction is added or modified.

The update history structure may be implemented using an unreplicated attribute of each log database object, an update tracking object in the log database, or other means. The update history structure is indexed on the UOID of the target database object it refers to and contains as an attribute the UOID of the most recent previous update of the target database object in the log database. In an alternative embodiment, the update history structure is implemented using a field or attribute in the target database rather than the log database.

In one embodiment according to the accessing step 116, each update object in the transaction log has one or more unreplicated attributes containing the UOID of the previous update object affecting the same database object. In addition, there exist objects indexed on the UOID of the database object whose history is concerned; these objects contain the UOID of the most recent log update object affecting the database object in question and act as chain headers. For efficiency, several of these headers may be combined into a single object. The chain is doubly linked to support log management steps that are best implemented through chaining in either direction. For efficiency, three dependency relationships are separately tracked:

i) Modification to the object in question itself and naming modifications (renames/moves) of its children;

ii) Modifications to the object's parent or to the parent's naming; and

iii) Modifications to the object's old parent or to the old parent's naming (move-performing updates only).

Such separate tracking makes it possible to track naming changes (renames and moves) in order to identify incompressible sequences in step 106. Move operations are effectively naming changes in both source and parent directories, so move-performing updates go onto three chains. If object naming changes (through renames) were kept on the same chain as other object updates, then coupling of dependency chains through renames and moves could cause all dependencies to degenerate into one long chain which provides no benefit because it would be equivalent to linearly scanning the log in reverse order. Accordingly, separate tracking is utilized.

As noted, each completed transaction in the transaction log has a corresponding transaction sequence number. The transaction sequence numbers are consecutive and monotonic for all completed transactions. The transaction numbers are stored in transaction sequence objects in the log database. By specifying a range of one or more transaction sequence numbers, the replica manager 46 can retrieve transactions from the transaction log in order according to their respective transaction sequence numbers during a reading step 122.

One method of the present invention uses the transaction identifying step 104, the update history structure accessing step 116, and a constructing step 124 to provide a prior version of a target database object. More particularly, a list of attributes is constructed representing the attributes which

have changed since the requested point in transaction history; for each of these attributes an earlier value is required.

Initially this list of required attributes is empty. The list is populated as follows. During a locating step, the most recent updating update for the object in question is found using the dependency chain header indexed by the object's UOID. A testing step tests whether the update found is earlier than the specified point in transaction history. If it is, execution skips to an iterating step described below. Otherwise, any attributes modified by the update found are added to the attribute list, the previous update affecting the same object is found using the dependency chains, and execution loops back to the testing step.

After the list is populated, the iterating step is performed. The replica manager 46 iterates backwards along the object's dependency chain from the point reached using the dependency chains, and for each update visited (including the one the iteration begins from) checks the populated attribute list. If the update modifies a listed attribute the replica manager 46 adds the modified value to the list and marks that attribute as no longer required. When no more attribute values are required, the iteration stops. The attribute list then represents changes which are applied to the current version of the object to reconstruct the historical version.

Thus, during one or more iterations of a modifying step 118, and a removing step 120, the log database is managed to achieve the steps shown in Figure 4. Operations on the updates are accomplished by corresponding operations on the linear log on disk or on update objects in the log database. In one

embodiment, functions are provided in the transaction logger 88 as follows:

object inserting step 114	tl_append() and/or tl_insert()
object removing step 120	tl_remove_record()
object modifying step 118	tl_remap_update_target(), tl_remap_uoid()
accessing step 116, identifying step 104, constructing step 124	tl_read_historical_object()
reading step 122	tl_readn()
identifying step 106	depend_update_dependent()
steps 108, 110, 102	various compression functions

In summary, the present invention provides a system and method for compressing a log of transactions performed on disconnectable computers. Redundant updates in the transaction log are identified through semantic tests and then removed. Operations are performed either directly on a disk-based log or by manipulation of objects and attributes in a log database. The present invention is well suited for use with systems and methods for transaction synchronization because the invention is implemented using replica managers 46 that perform synchronization and the log compression steps of the present invention may be used to remove transient clashes that arise during synchronization. The architecture of the present invention is not limited to file system operations but can instead be extended to support a variety of target and/or log database objects.

Although particular methods embodying the present invention are expressly illustrated and described herein, it will be appreciated that apparatus and article embodiments may be formed according to methods of the present invention. Unless otherwise expressly indicated, the description herein of methods of the present invention therefore extends to

corresponding apparatus and articles, and the description of apparatus and articles of the present invention extends likewise to corresponding methods.

The invention may be embodied in other specific forms without departing from its essential characteristics. The described embodiments are to be considered in all respects only as illustrative and not restrictive. Any explanations provided herein of the scientific principles employed in the present invention are illustrative only. The scope of the invention is, therefore, indicated by the appended claims rather than by the foregoing description. All changes which come within the meaning and range of equivalency of the claims are to be embraced within their scope.

What is claimed and desired to be secured by patent is:

CLAIMS

1. A method for managing a transaction log, the log representing a sequence of transactions in a network of connectable computers, each transaction containing at least one update targeting a target database object in a distributed hierarchical target database that contains convergently consistent replicas residing on separate computers in the network, said method comprising the computer-implemented steps of identifying at least one redundant update in the transaction log and then removing the redundant update from the transaction log.

2. The method of claim 1, further comprising the computer-implemented step of identifying an incompressible sequence of updates in the transaction log.

3. The method of claim 1, further comprising the computer-implemented step of identifying a transaction boundary within the transaction log.

4. The method of claim 3, wherein said step of removing the redundant update comprises the steps of determining the most recent successfully merged transaction that updates a selected object and then removing an update of the object that antedates the transaction.

5. The method of claim 1, wherein the transaction log resides on a first computer and said step of removing the redundant update comprises the steps of:

identifying an update in the transaction log that specifies an update location on a computer other than the first computer; and then

removing that update.

6. The method of claim 1, wherein said step of removing the redundant update comprises the steps of:

repositioning an update in the sequence of updates in the transaction log; and then

replacing the repositioned update and an adjacent update by a single equivalent update.

7. The method of claim 1, wherein the network includes a server computer and a client computer, a server replica of the target database resides on the server computer, and a client replica of the target database resides on the client computer.

8. The method of claim 1, wherein said identifying and removing steps are preceded by the computer-implemented step of creating a hierarchical log database representing at least a specified portion of the transaction log, the log database containing an object corresponding to an update in the specified portion and also containing an object corresponding to a transaction in the specified portion of the transaction log.

9. The method of claim 8, wherein the specified portion of the transaction log is the entire existing transaction log.

10. The method of claim 8, wherein the specified portion of the transaction log includes recent transactions and the remainder of the existing transaction log is represented by a compressed linearly accessed log.

11. The method of claim 8, wherein said method further comprises the computer-implemented step of appending a

transaction to the transaction log by inserting a transaction object into the log database.

12. The method of claim 11, wherein said appending step comprises inserting an update object into the log database.

13. The method of claim 11, wherein said appending step comprises accessing an unreplicated attribute in the log database to identify an earlier update, if any, which references an object in the target database that is also referenced by an update in the appended transaction.

14. The method of claim 11, wherein said appending step comprises accessing an update history structure in the log database to identify an earlier update, if any, which references an object in the target database that is also referenced by an update in the appended transaction, the update history structure associating each target database object with the log database objects, if any, that correspond to updates referencing the given target database object.

15. The method of claim 8, wherein said method further comprises the computer-implemented step of adding a synchronization checkpoint to the transaction log by inserting a synchronization checkpoint object into the log database.

16. The method of claim 8, wherein said method further comprises the computer-implemented steps of removing a synchronization checkpoint from the transaction log by removing a synchronization checkpoint object from the log database and then compressing a previously incompressible region of the transaction log.

17. The method of claim 8, wherein each completed transaction in the transaction log has a corresponding transaction sequence number, the transaction sequence numbers are consecutive and monotonic for all completed transactions, and said method further comprises the computer-implemented steps of specifying a range of one or more transaction sequence numbers and then retrieving transactions from the transaction log in order according to their respective transaction sequence numbers.

18. A method for managing a transaction log, the log representing a sequence of transactions in a network of connectable computers, each transaction containing at least one update targeting a target database object in a distributed hierarchical target database that contains convergently consistent replicas residing on separate computers in the network, said method comprising the computer-implemented steps of:

creating a log database representing at least a specified portion of the transaction log, the log database containing a log database object corresponding to an update in the specified portion and also containing a log database object corresponding to a transaction in the specified portion of the transaction log; and

creating an update history structure which associates each target database object with the log database objects, if any, that correspond to updates referencing the given target database object.

19. The method of claim 18, further comprising the computer-implemented steps of locating a transaction checkpoint, accessing the update history structure, and then constructing a prior version of a target database object.

20. A computer-readable storage medium having a configuration that represents data and instructions which cause a disconnectable computer to perform method steps for managing a transaction log, the log representing a sequence of transactions in a network of connectable computers, each transaction containing at least one update targeting a target database object in a distributed target database that contains convergently consistent replicas residing on separate computers in the network, the method comprising the computer-implemented steps of identifying at least one redundant update in the transaction log and then removing the redundant update from the transaction log.

21. The storage medium of claim 20, wherein the method further comprises the computer-implemented step of identifying an incompressible sequence of updates in the transaction log.

22. The storage medium of claim 20, wherein the method further comprises the computer-implemented step of identifying a transaction boundary within the transaction log.

23. The storage medium of claim 20, wherein the step of removing the redundant update comprises the steps of repositioning an update in the sequence of updates in the transaction log and then replacing the repositioned update and an adjacent update by a single equivalent update.

24. The storage medium of claim 20, wherein the identifying and removing steps are preceded by the computer-implemented step of creating a hierarchical log database representing at least a specified portion of the transaction log, the log database containing an object corresponding to an update in the specified portion and also containing an object corresponding to a transaction in the specified portion of the transaction log.

25. The storage medium of claim 24, wherein the specified portion of the transaction log includes recent transactions and the remainder of the existing transaction log is represented by a compressed linearly accessed log.

26. The storage medium of claim 24, wherein the method further comprises the computer-implemented step of extending the transaction log by inserting an object into the log database.

27. The storage medium of claim 24, wherein the method further comprises the computer-implemented step of accessing an update history structure in the log database to identify an earlier update, if any, which references an object in the target database that is also referenced by an update in the appended transaction.

28. The storage medium of claim 27, wherein the method further comprises the computer-implemented steps of locating a transaction checkpoint and constructing a prior version of a target database object.

29. The storage medium of claim 24, wherein each completed transaction in the transaction log has a corresponding transaction sequence number, and the transaction

sequence numbers are consecutive and monotonic for all completed transactions.

30. A system for managing a transaction log, the log representing a sequence of transactions in a network of connectable computers, each transaction containing at least one update targeting a target database object in a distributed target database that contains convergently consistent replicas residing on separate computers in the network, said system comprising a computer comprising means for storing the log and means for executing programmed instructions, means for identifying at least one redundant update in the transaction log, and means for removing the redundant update from the transaction log.

31. The system of claim 30, further comprising means for identifying an incompressible sequence of updates in the transaction log.

32. The system of claim 30, further comprising means for identifying a transaction boundary within the transaction log.

33. The system of claim 30, further comprising means for creating a hierarchical log database representing at least a specified portion of the transaction log, the log database containing an object corresponding to an update in the specified portion and also containing an object corresponding to a transaction in the specified portion of the transaction log.

34. The system of claim 30, wherein said means for removing the redundant update comprises means for repositioning an update in the sequence of updates in the transaction log and

means for replacing the repositioned update and an adjacent update by a single equivalent update.

35. The system of claim 33, wherein the specified portion of the transaction log includes recent transactions and the remainder of the existing transaction log is represented by a compressed linearly accessed log.

36. The system of claim 33, further comprising means for extending the transaction log by inserting an object into the log database.

37. The system of claim 33, further comprising means for accessing an update history structure in the log database to identify an earlier update, if any, which references an object in the target database that is also referenced by an update in the appended transaction.

38. The system of claim 37, further comprising means for constructing a prior version of a target database object.

39. The system of claim 33, further comprising means for removing a synchronization checkpoint from the transaction log and means for compressing a previously incompressible region of the transaction log.

40. The system of claim 33, wherein each completed transaction in the transaction log has a corresponding transaction sequence number, and the transaction sequence numbers are consecutive and monotonic for all completed transactions.

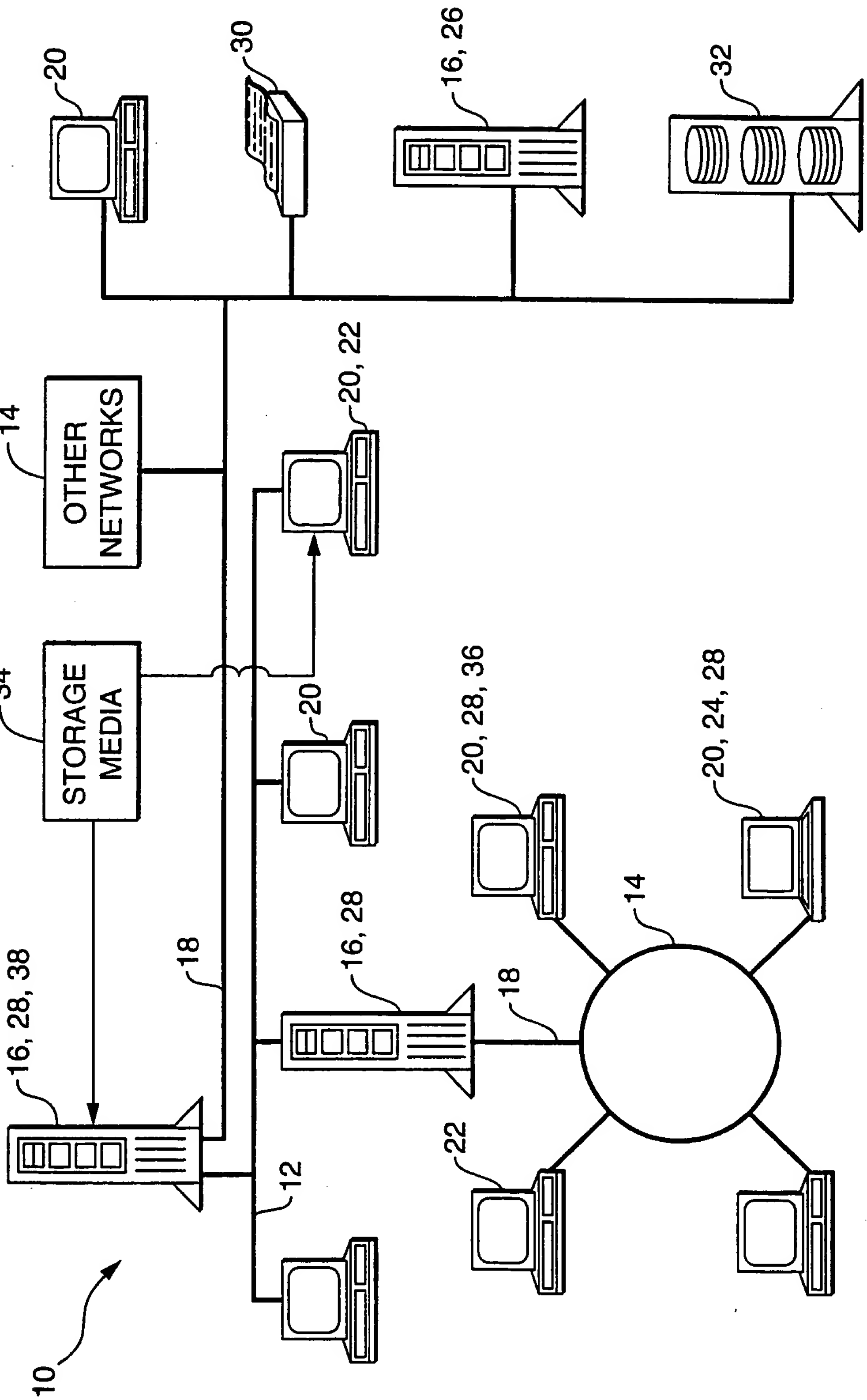
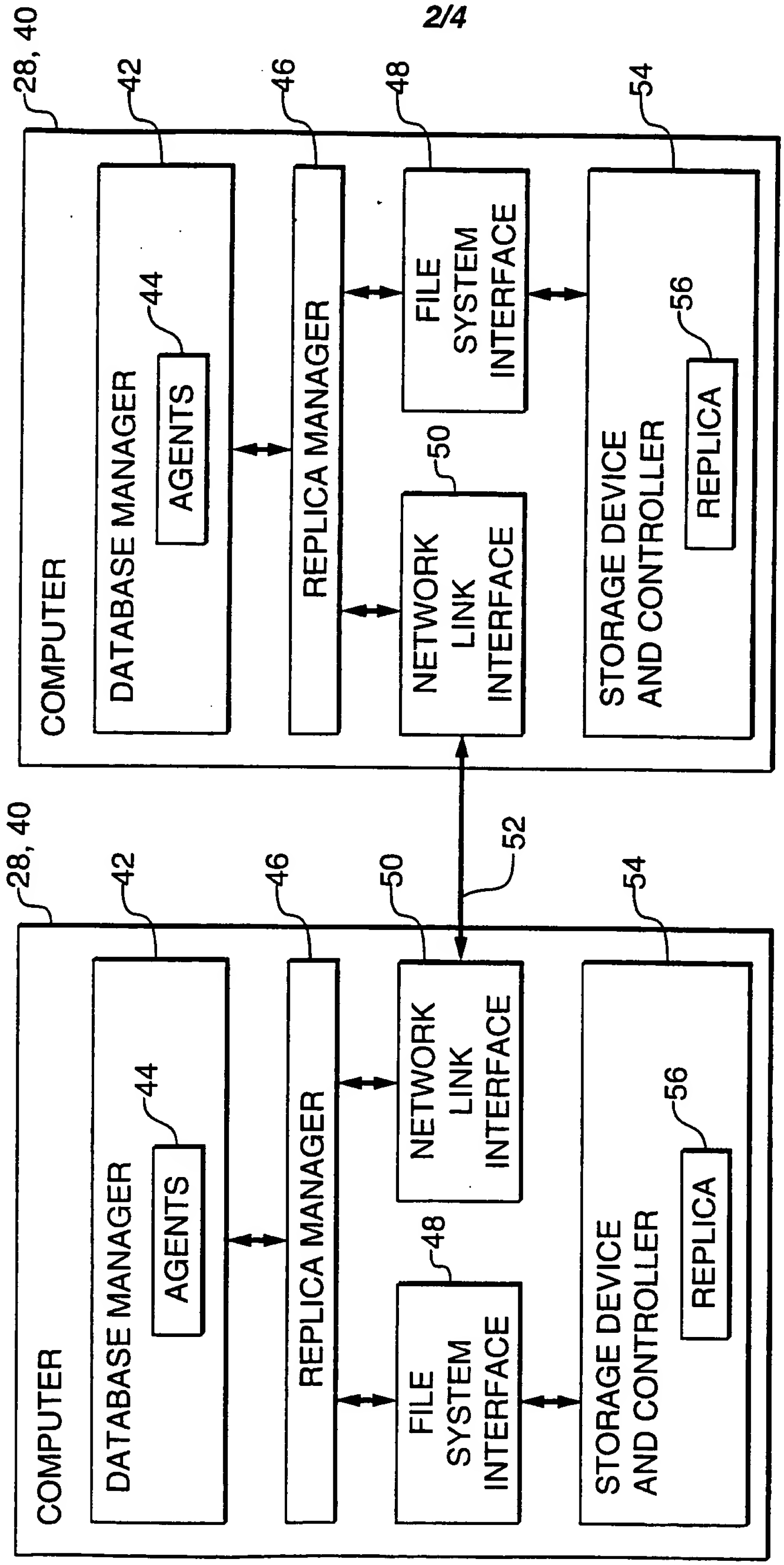

10
16, 28, 38
18
12
STORAGE MEDIA
34
OTHER NETWORKS
14
20
30
16, 26
32
20, 22
16, 28
18
14
22
20, 28, 36
20, 24, 28

FIG. 1



COMPUTER
DATABASE MANAGER
AGENTS
REPLICA MANAGER
FILE SYSTEM INTERFACE
NETWORK LINK INTERFACE
STORAGE DEVICE AND CONTROLLER
REPLICA
28, 40
42
44
46
48
50
52
54
56

FIG. 2

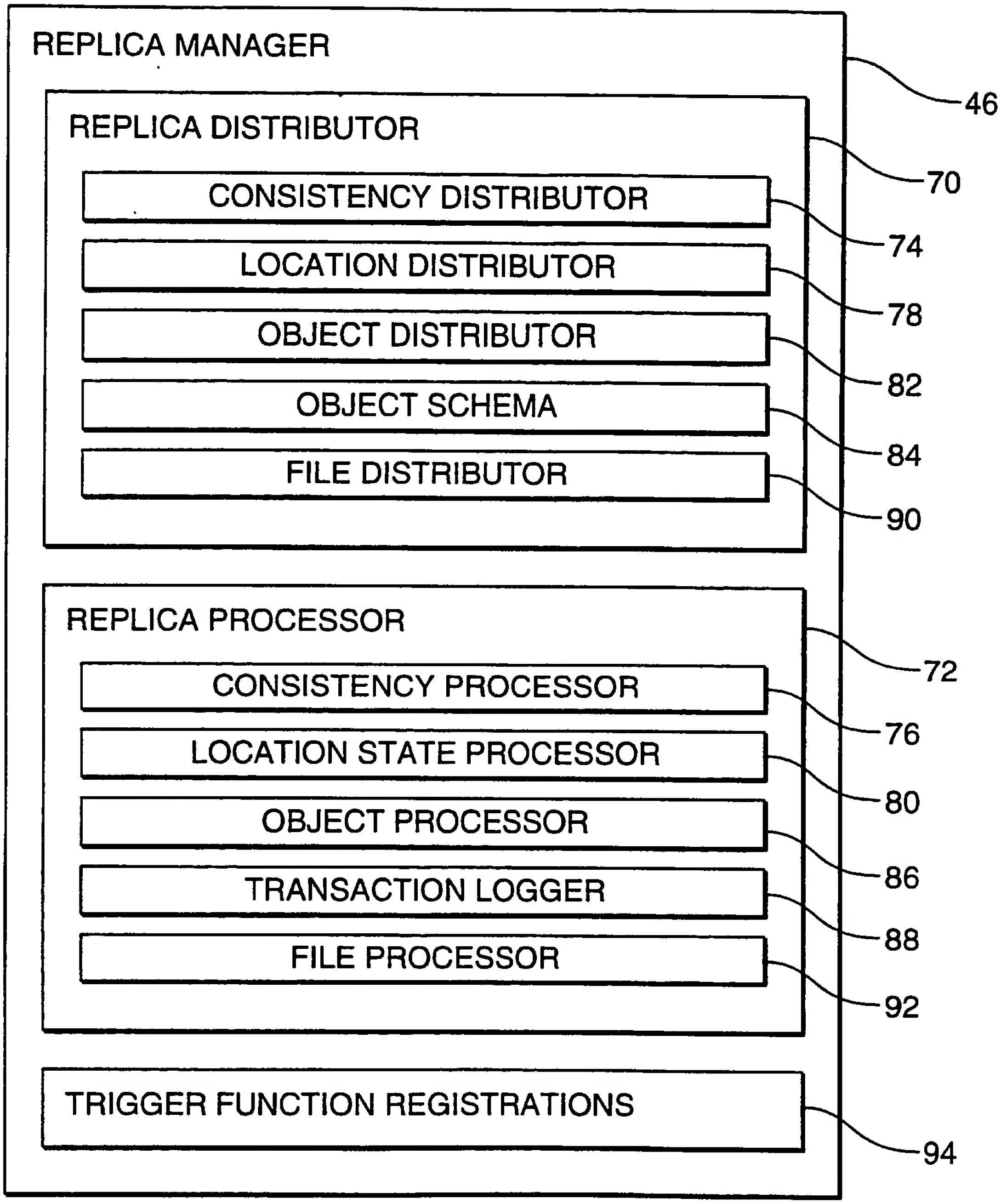

REPLICA MANAGER
REPLICA DISTRIBUTOR
CONSISTENCY DISTRIBUTOR
LOCATION DISTRIBUTOR
OBJECT DISTRIBUTOR
OBJECT SCHEMA
FILE DISTRIBUTOR
REPLICA PROCESSOR
CONSISTENCY PROCESSOR
LOCATION STATE PROCESSOR
OBJECT PROCESSOR
TRANSACTION LOGGER
FILE PROCESSOR
TRIGGER FUNCTION REGISTRATIONS
46
70
74
78
82
84
90
72
76
80
86
88
92
94

FIG. 3

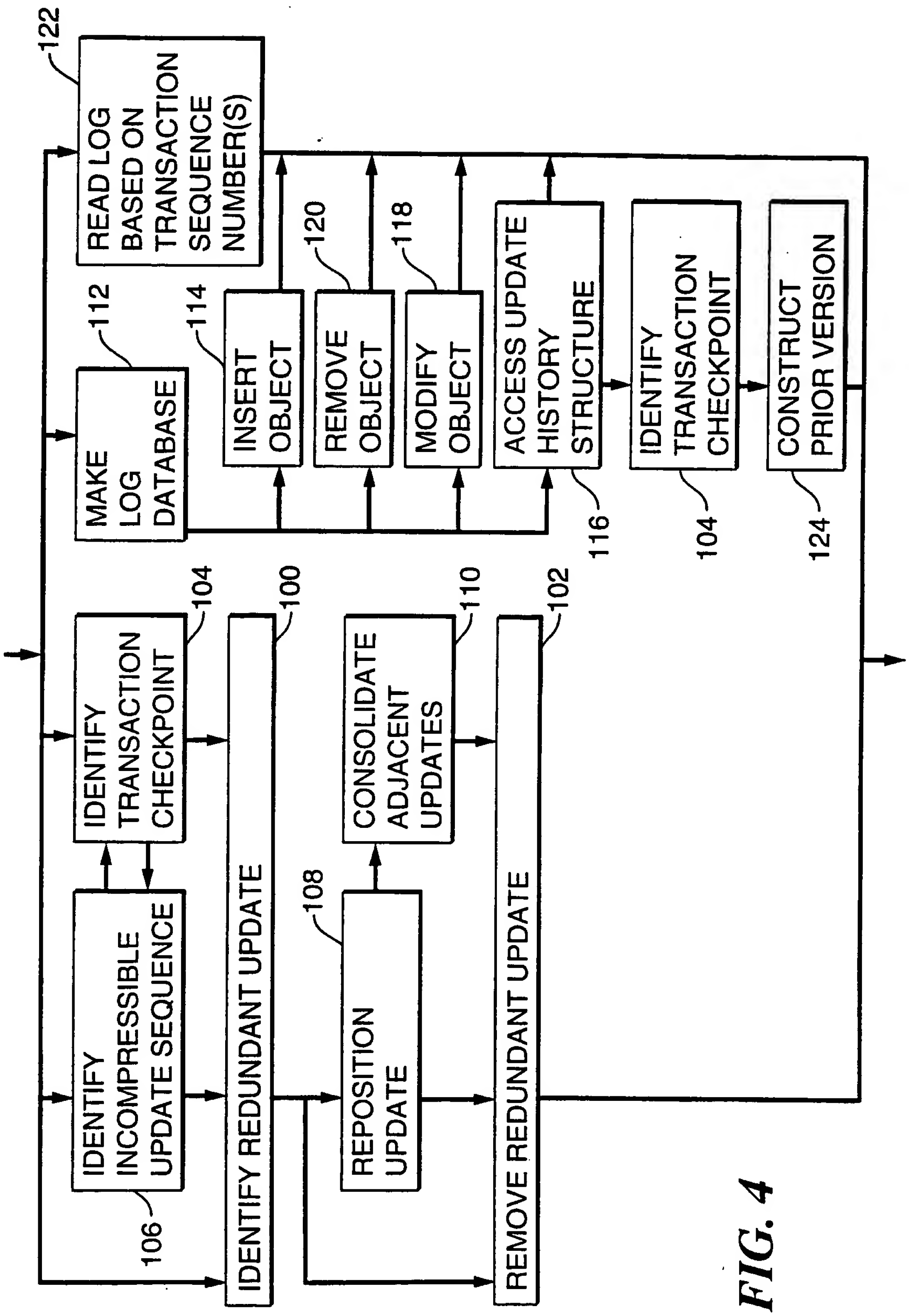

IDENTIFY TRANSACTION CHECKPOINT
104
IDENTIFY INCOMPRESSIBLE UPDATE SEQUENCE
106
IDENTIFY REDUNDANT UPDATE
100
REPOSITION UPDATE
108
CONSOLIDATE ADJACENT UPDATES
110
REMOVE REDUNDANT UPDATE
102
MAKE LOG DATABASE
112
READ LOG BASED ON TRANSACTION SEQUENCE NUMBER(S)
122
INSERT OBJECT
114
REMOVE OBJECT
120
MODIFY OBJECT
118
ACCESS UPDATE HISTORY STRUCTURE
116
IDENTIFY TRANSACTION CHECKPOINT
104
CONSTRUCT PRIOR VERSION
124

FIG. 4

INTERNATIONAL SEARCH REPORT

International Application No
PCT/US 96/11903

A. CLASSIFICATION OF SUBJECT MATTER
IPC 6 G06F11/14 G06F9/46 G06F17/30

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 6 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X A	C.J. DATE: "An Introduction to Database Systems, Volume II" July 1985 , ADDISON-WESLEY PUBLISHING COMPANY , READING, MA, US XP002016220 pages 1-33 (Chapter 1); pages 291-340 (Chapter 7) see page 13, line 1 - line 14 see page 291, line 1 - page 295, line 20 see page 306, line 34 - page 309, line 26 ---	1,20,30 2-6,18, 21-23, 31,32
A	EP,A,0 250 847 (IBM) 7 January 1988 see abstract see page 3, line 23 - page 4, line 10 see claim 1 ---	1,20,30
	-/--	

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

1

Date of the actual completion of the international search	Date of mailing of the international search report
18 October 1996	04.11.96
Name and mailing address of the ISA European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Authorized officer Wiltink, J

INTERNATIONAL SEARCH REPORT

In tional Application No
PCT/US 96/11903

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	ASSOCIATION FOR COMPUTING MACHINERY SPECIAL INTEREST GROUP ON MANAGEMENT OF DATA (SIGMOD) 1987 ANNUAL CONFERENCE, vol. 16, no. 3, 27 - 29 May 1987, ISSN 0163-5808, SAN FRANCISCO, CA, USA, pages 82-96, XP002016376 DEAN S. DANIELS ET AL.: "Distributed Logging for Transaction Processing" -----	1,5,7, 20,30

1

INTERNATIONAL SEARCH REPORT

Information on patent family members

International Application No
PCT/US 96/11903

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
EP-A-0250847	07-01-88	US-A- 4878167	31-10-89
		DE-A- 3786956	16-09-93
		DE-T- 3786956	17-03-94
		JP-C- 1856908	07-07-94
		JP-A- 63010252	16-01-88